Summer 8-15-2021

Provably and Practically Efficient Race Detection for Task-Parallel Code

Yifan Xu

WASHINGTON UNIVERSITY IN ST.LOUIS

McKelvey School of Engineering
Department of Computer Science and Engineering

Dissertation Examination Committee:
I-Ting Angelina Lee, Chair
Kunal Agrawal
Sanjoy Baruah
Jeremy Buhler
Jeremy T. Fineman (Georgetown)

Provably and Practically Efficient Race Detection for Task-Parallel Code

by

Yifan Xu

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2021
St. Louis, Missouri

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgments

Every story has an ending, just as I am writing these acknowledgments to conclude my PhD life. Five years ago, I left my hometown, Shanghai, and began studying abroad for the first time in my life. Before I came to WashU, I had negligible research experience. I did not know if I could take care of myself without my parents' help. I did not know how to drive. There were a lot of things I needed to learn to survive my graduate study. Therefore, I was imagining a difficult and painful process of pursuing my PhD that I might not be able to stand. Every single moment from the past five years, however, had been proving that I was wrong. I published my first paper after the first year. I attended an academic conference for the first time and met interesting people. I have had a lot of such happy memories during my graduate study, and none of them would have been possible without the help of many people.

First and foremost, I would like to thank my advisor, Dr. Angelina Lee. Angelina is truly a wonderful advisor. Her advice is always very helpful to me in finding the right problem to work on. She has also taught me to think critically when reading papers and encouraged me to develop my own ideas, which benefit not only my PhD study but also my future career. I still remember that we worked together until early morning, fighting my first paper deadline. This experience sounds painful, but to me, it is truly a wonderful memory. I am very glad that she received tenure recently.

I would like to thank my reliable collaborators, especially Dr. Kunal Agrawal and my lab mate, Kyle Singer. Kunal is a great theory person, and her insights on algorithms really make my research process smooth. She and Angelina together have also helped me to improve my presentation skills through the iterations of conference talk practice.

Kyle joined our group in 2017. He has brilliant abilities to build reliable and efficient runtime systems. I cannot have most of my work done without his support.

I want to thank the other members who serve on my committee: Dr. Sanjoy Baruah, Dr. Jeremy Buhler and Dr. Jeremy Fineman. Especially, I am very grateful to Dr. Fineman for his attendance at my defense virtually from Sydney after midnight.

I am fortunate to have these great friends during my graduate study at WashU: Ruixuan Dai, Shenghua He, Dingwen Li, Songshan Liu, Wei Tang, Xiaojian Xu, Hao Yan, and Huayi Zeng. We have had a lot of fun together, including working out in the gym, hiking, going to Karaoke, having hotpot, so on and so forth, which really makes my life here colorful. I would also like to thank all my friends in Shanghai, especially Weiwei Cai, Fan Gu, Merry Hou, Zhengzhen Huang, Liang Liang, Yong Sun and Yuanyuan Zhu, for often chatting with me and sharing me interesting news from my hometown.

I would like to thank my lovely girlfriend, Tiantian Zhu, for appearing in my life. Tiantian is the most shining girl I have ever met. Even if she is not a Computer Science person, she is always willing to listen to my complaints about the difficulties I met during my research. I am also thankful for her great cooking skills, which allow me to have had so much delicious Chinese food in St. Louis where there are not many good Chinese restaurants.

I would like to end this list of acknowledgments by showing my deepest gratefulness to my Dad and Mom, for having taught me everything that cannot be taught from schools, especially, to be a nice and kind person.

The work described in this dissertation is supported in part by the National Science Foundation under grant numbers CCF-1527692, CCF-1150036, CCF-1733873, CCF-1910568,

CCF-1943456, CCF-1533644, CCF-1725647 and CCF-1439062; and by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000.

Yifan Xu

Washington University in St. Louis

December 2021

Dedicated to my parents,

Xu Jiu and Xu Ruiping,

who have encouraged me to begin this journey.

ABSTRACT OF THE DISSERTATION

Provably and Practically Efficient Race Detection for Task-Parallel Code

by

Yifan Xu

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2021

Advisor: I-Ting Angelina Lee

Parallel systems are pervasive nowadays. Specifically, modern computers have embraced multicore architectures due to the difficulties of exploiting higher clock speeds on single-core CPUs. However, parallel programming is challenging. Determinacy race, in particular, is a common pitfall when writing task-parallel code. It can easily lead to non-deterministic behavior of the parallel program and therefore a determinacy race is often considered as a bug. Unfortunately, such bugs are hard to debug because they do not necessarily produce obvious failures in every single execution.

To ease the debugging process of determinacy races in task-parallel code, this dissertation proposes several provably and practically efficient parallel race detection algorithms. Unlike prior works mostly target fork-join parallelism, we focus on less structured but important programming paradigms – pipeline parallelism and futures. In addition, we build an efficient runtime system for scheduling futures, which is not only a facility to study the race detection problem for futures but also useful in practice. Finally, this dissertation investigates mechanisms that optimize the access history of race detectors, which provides significant additional boost to the performance.

Chapter 1

Introduction

Parallel systems are pervasive nowadays. Specifically, modern computers have embraced multicore architectures due to the difficulties of exploiting higher clock speeds on single-core CPUs. Unlike rising clock speeds, serial programs are not getting much performance boost through the parallel architectures. Scalable parallel programs, in contrast, can make full use of the powerful capabilities of multicore CPUs.

However, parallelization of applications can't be done automatically. Programmers are now required to write correct and scalable parallel programs explicitly despite the fact that the serial programming abstraction was widely used before. Worse yet, parallel programming is inherently more challenging than serial programming. For example, when programming with low-level threading APIs, such as POSIX thread [40], programmers need to manually manage task decomposition, scheduling and complex synchronization.

Fortunately, much effort has been made to simplify parallel programming. Task parallelism, for example, provides programmers with language constructs to express logical parallelism of the computation, and an underlying runtime system that performs the scheduling

and synchronization among parallel computations to achieve load balancing and coordination. Such high-level programming model allows programmers to focus on the decomposition of the problem and the design of the parallel algorithm.

1.1 Pitfall: Determinacy Race

Even though task parallelism meets a wide range of parallel programming needs, parallel programming still remains challenging for programmers. Unlike serial programs that are naturally deterministic, meaning that the program will always produce the same output given a particular input, parallel programs can easily become non-deterministic. Take the simple task-parallel code in Figure 1.1 as an example. The `spawn` keyword in the main function indicates that `task1` can be executed concurrently with the continuation in which `task2` is invoked (lines 9–10). Then the `sync` keyword causes the main function to suspend until `task1` returns (line 11). A risk in this task-parallel code is that, `task1` and `task2` update the shared variable `x` with different values concurrently. Therefore the output value of `x` can vary, depending on the scheduling order.

```
1  int x;
2  void task1() {
3      x = 1;
4  }
5  void task2() {
6      x = 2;
7  }
8  int main() {
9      spawn task1();
10     task2();
11     sync;
12     printf("x is %d\n", x);
13     return 0;
14 }
```

Figure 1.1: A simple task-parallel code that contains determinacy races.

Such risk is commonly referred to as determinacy race. A ***determinacy race*** occurs when two or more logically parallel tasks perform memory operations on the same memory location and one of the operations is a write. [1] Determinacy races can lead to nondeterministic behavior and therefore they are often bugs. Unfortunately, such bugs are hard to debug because they do not necessarily produce obvious failures in every single execution.

To that end, many race detection algorithms [2, 9, 27–29, 55, 71, 72, 90, 97, 98] have been proposed in the context of task parallelism. The algorithms usually perform race detection on-the-fly as the program executes. For a given program and input, the race detection algorithms report a race if and only if the program contains a determinacy race for that input, regardless of the schedule. Moreover, the algorithms typically consist of two components: an ***access history*** that keeps track of the readers and the writers that previously accessed a given memory location during execution and a ***reachability data structure*** that, given a reader and a writer that accessed the same memory location, answers the question of whether or not they are logically in parallel.

1.2 Limitations of the Prior Studies

Each of the prior works has its own limitation. First, much of the prior work [9, 27–29, 55, 71, 72, 97] supports race detecting only fork-join parallelism (formally defined in Section 2.1). Fork-join parallelism has nice structural properties and such properties enable efficient race detection algorithms. A few prior works [2, 26, 90, 98] have studied the problems outside of the realm of fork-join parallelism. Due to the lack of structural properties, however, the algorithms execute the computation sequentially and incur a large overhead. Second, most of the work has been done focused on optimizing the reachability while little attention has been paid to the access history. It is often the most expensive component of race detection in practice while in theory, the access history adds no asymptotic overhead.

[1] In contrast, a ***data race*** occurs when two parallel strands, holding no locks in common, access the same memory location in a conflicting way. A determinacy race is sometimes referred to as a ***general race*** [61].

1.3 Contributions

This dissertation, therefore, proposes several provably and practically efficient parallel race detection algorithms that detect determinacy races for computations that are less structured than fork-join parallelism. In addition, we have investigated mechanisms that optimize the access history component in race detection algorithms. The rest of this chapter briefly summarizes the contributions of this dissertation.

Race detection for pipeline parallelism

We propose a provably correct and efficient race detection algorithm, ***2D-Order***, for detecting races in pipeline parallelism. 2D-Order is the first known parallel race detection algorithm that targets pipeline program and given a computation, 2D-Order executes it while also detecting races with asymptotically constant overhead. We also implemented PRacer, a race detector based on 2D-Order for Cilk-P [48, 49], which is a language for expressing pipeline parallelism. Empirical results demonstrate that 2D-Order incurs reasonable overhead and exhibits scalability similar to the baseline (executions without race detection) when running on multiple cores.

Results of this study have previously appeared in the following publication:

- Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal, Efficient Parallel Determinacy Race Detection for Two-Dimensional Dags, *Proceedings of the 23rd Symposium on Principles and Practice of Parallel Programming* (PPoPP'18), 2018.

Race detection for future parallelism

A provably and practically efficient scheduler for future parallelism is needed when studying the race detection problem in the context of futures. However, a program with futures could incur much higher scheduling costs, when scheduling with the ***classic work-stealing*** algorithm (described in Section 2.3). Therefore, we first investigate an alternative scheduling

approach, called ***proactive work-stealing***, an algorithm for scheduling the future parallelism with provably efficient execution time and better cache performance compared to classic work-stealing.

We then address race detection problems for programs with different use of futures, namely ***general futures*** and ***structured futures***. Specifically, we present two algorithms: ***F-Order*** and ***SF-Oder***. F-Order is the first known parallel race detection algorithm that detects races on programs using general futures. SF-Order, in contrast, exploits the restrictions imposed by structured futures, and therefore achieves better execution time compared to the race detection algorithm designed for general futures. We implemented both algorithms and empirically demonstrated their efficiency.

Results of this work have previously appeared in the following publications:

- Kyle Singer, Yifan Xu, and I-Ting Angelina Lee, Proactive Work Stealing for Futures, *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (PPoPP'19), 2019.

- Yifan Xu, Kyle Singer, and I-Ting Angelina Lee, Parallel Determinacy Race Detection for Futures, *Proceedings of the 25th Symposium on Principles and Practice of Parallel Programming* (PPoPP'20), 2020.

- Yifan Xu, Kunal Agrawal, and I-Ting Angelina Lee, Efficient Parallel Determinacy Race Detection for Structured Futures, *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA'21), 2021.

Optimizing access history

We propose compiler and runtime mechanisms that perform memory access coalescing efficiently, and a treap-based access history data structure. Together these optimization are capable of speeding up sequential race detectors for task-parallel code.

We then extend those optimization of access history to parallel race detectors by exploring an ***asynchronous access history*** scheme.

Results of part of this work have previously appeared in the following publication:

- Yifan Xu, Anchengcheng Zhou, Grace Q. Yin, Kunal Agrawal, I-Ting Angelina Lee, and Tao B. Schardl, Brief Announcement: Efficient Access History for Race Detection, *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA'21), 2021.

1.4 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 reviews the backgrounds and introduces the terminologies used throughout this dissertation. Chapter 3 addresses the race detection problem for pipeline parallelism. Chapter 4 introduces future parallelism and presents proactive work-stealing – the scheduling algorithm that serves as an essential basis for the works in the following chapters. Chapter 5 and Chapter 6 propose race detection algorithms for general futures and structured futures, respectively. Chapter 7 investigates optimizations on access history to speed up sequential race detectors and Chapter 8 extends those optimizations to parallel race detectors. Chapter 9 reviews the related work. Chapter 10 concludes this dissertation by summarizing the results presented throughout the document.

Chapter 2

Preliminary

2.1 Task Parallelism

Task parallelism is a processor-oblivious programming model that provides programmers language constructs to express logical parallelism of the computation. During the execution of a task-parallel program, an underlying runtime scheduler automates tasks such as load balancing and coordination among parallel tasks. Task parallelism is widely supported by parallel programming platforms such as Intel TBB [43], Cilk dialects [24, 44, 50], Habanero dialects [7, 16] and X10 [18].

Based on the sets of language constructs used in the task-parallel program, one can divide task parallelism into categories. Take ***fork-join***, a traditional parallelism paradigm supported by many task-parallel platforms, for example. Fork-join parallelism can be expressed using two simple keywords: `spawn` and `sync`. When a function F spawns off another function G by prefixing the invocation with `spawn`, the execution of G may operate in parallel with the continuation of F. The invocation of a `sync` specifies that all previously

spawned functions must return before the control can pass `sync`.[2] Figure 2.1 shows a simple fork-join code that computes the nth Fibonacci number in a naive way.

```
1 int fib(int n) {
2     if (n < 2) return n;
3     int x = spawn fib(n-1);
4     int y = fib(n-2);
5     sync;
6     return x + y;
7 }
```

Figure 2.1: A simple fork-join code that computes the nth Fibonacci number.

2.2 Modeling Parallel Computations

One can model the execution of a parallel program as a ***directed acyclic graph (dag)***, where a node represents a ***strand*** or a series of sequential instructions without parallel constructs, and an edge represents a dependence between two strands. Given two nodes u and v such that there exists a path from u to v, for example, then the strand represented by u must be completed before executing the strand represented by v. In other words, the incidence relations of a dag can be viewed as order relations on its nodes and edges. Notation-wise, we write $u \rightsquigarrow v$ to denote the presence of a directed path from u to v. We say $u \prec v$ iff $u \rightsquigarrow v$; $u \preceq v$ iff either $u \prec v$ or $u = v$. We say $u \parallel v$ iff there is no path from u to v or from v to u.

Given a computation dag, one can measure its performance using two metrics: the ***work*** metric, which is the number of nodes in the dag, and the ***span*** metric, which is the length of the longest directed path through the dag, assuming each node takes unit time to execute. Equivalently, the work metric measures how long it takes to run this computation on one

[2]Many task-parallel platforms also support parallel loops, which can be thought of as syntactic sugar that compiles down to `spawn` and `sync`.

core and the span metric measures how long it takes to run this computation on an infinite number of cores.

Series-parallel dag

The execution of a fork-join program generates a ***series-parallel dag (SP-dag)*** with well-defined structural properties. An SP-dag is a planar dag with a unique source node with no incoming edges and a unique sink node with no outgoing edges. Specifically, the execution of a `spawn` creates a ***spawn node*** with two outgoing edges: one to the spawned function and one to the continuation of the caller. The execution of a `sync` creates a ***sync node*** that joins together all previously spawned functions — creating an edge from the end of a spawned function to the sync node that represents continuation after the `sync` statement. Without loss of generality, we shall assume that a sync node consists only only two incoming edges — it is not difficult to convert a sync node with multiple incoming edges into a chain of sync nodes, each with two incoming edges.

2.3 Work-Stealing Scheduler

A task-parallel computation can be scheduled efficiently using a work-stealing scheduler [4, 5, 13, 14]. A ***work-stealing*** scheduler dynamically load balances a parallel computation in a distributed fashion, which incurs low contention compared to centralized approaches (e.g., work-sharing). In classic work-stealing, each ***worker*** thread keeps its own double-ended queue (or deque for short) which holds ***ready nodes***, that is, unexecuted nodes whose predecessors in the dag have all been executed. Each worker operates on its own deque locally. Specifically, when a worker executes a ready node in the dag and causes some child nodes to be ready, the worker pushes those ready nodes onto the bottom of its deque. Then the worker gets a new ready node by popping one off the bottom of its deque and continues to execute it. If a worker finds that its deque is empty, then the worker becomes a thief.

A thief worker chooses a victim worker uniformly at random and attempts to steal the top node of the victim's deque.

The work-stealing algorithm has been shown to provide strong performance guarantees [4, 5, 13, 14]. Given a parallel computation with work T_1 and span T_∞, a work-stealing scheduler executes it on P processors in expected execution time $T_1/P + O(T_\infty)$.

2.4 Determinacy Race Detection

Parallel programming introduces additional pitfalls compared to serial programming. Programmers are required to specify the execution order between certain tasks to coordinate the parallel execution flow. Inappropriately defined orders can lead to concurrency bugs, e.g., races and deadlocks. This dissertation targets determinacy races. A ***determinacy race*** occurs when two or more logically parallel instructions access the same memory location, and at least one of them performs a write. Determinacy races may lead to non-deterministic behavior that is generally a programming error.

On-the-fly race detector

In this work, we focus on performing race detection on-the-fly as the program executes. For a given program and input, we want to report a race if and only if the program contains a race for that input. As mentioned in Chapter 1, an on-the-fly race detector maintains two key data structures. An access history that stores the readers and the writers that previously accessed a given memory location, and a reachability data structure that determines whether two given accesses are logically in parallel or not, or equivalently whether or not there is a directed path between the corresponding nodes in the dag. For an on-the-fly race detector, the reachability data structure must be updated incrementally as new nodes in the dag are revealed during the execution. The race detector manages the data structures during the execution of the program. Given a strand u that accesses memory location l, the race

detector performs a reachability query between u and each conflicting access stored in the access history that accessed memory location l. If u is in parallel with any of the conflicting accesses, the detector finds a race.

Race detection for fork-join programs

Recall from Section 2.2, a dag can be viewed as partial order relations on its nodes and edges. This leads to the definition of the ***order-dimension*** — the order-dimension of a dag G is the minimum number of total orders on G such that, given two nodes u and v in G, $u \prec v$ iff u precedes v in all the total orders. SP-dags have been proved to have order-dimension two. Consequently, Nudler and Rudolph [62] introduce ***English*** and ***Hebrew*** order, that is, two total orders of nodes in SP-dags, to determine if two nodes are logically in parallel. Later, Bender et al. [9] propose ***SP-Order***, the first race detection algorithm for SP-dags with asymptotically optimal sequential running time, using a pair of ***order-maintenance (OM)*** data structures to perform maintenance of English and Hebrew order of all revealed nodes. ***WSP-Order*** [97] parallelizes SP-Order by incorporating additional runtime support and achieves the optimal parallel executing time, that is, given an SP-dag with work T_1 and span T_∞, WSP-Order runs in time $O(T_1/P + T_\infty)$ on P cores.

Chapter 3

Race Detection for Pipeline Parallelism

Pipeline parallelism organizes a parallel program as a linear sequence of stages. Conceptually, each stage processes elements of a data stream and the stream flows through the pipeline from the first stage to the last stage. Pipeline parallelism is widely supported [3, 22, 35, 38, 52, 53, 64, 64, 68, 70, 73–76, 78, 89, 95], including Intel's Threading Building Blocks (TBB) [54], OpenMP [63], and Cilk-P [48, 49], an extension to Cilk designed specifically for linear pipelines. It has been shown that a program with pipeline parallelism can be scheduled efficiently using work-stealing [48, 49].

A pipeline program can be represented using a ***two-dimensional dag (2D-dag)***, that is , a planar directed acyclic graph that can be embedded within a two-dimensional grid space.[3] In this chapter, we present ***2D-Order***, a provably correct and efficient race detection algorithm for 2D-dags, that has an asymptotically optimal parallel running time. Given a pipeline program with work T_1 and span T_∞, 2D-Order can detect races while executing the computation on P processors with expected time $O(T_1/P + T_\infty)$. This bound is asymptotically optimal, since this is the best one can do when executing the same program without

[3] We formally define 2D-dags that our algorithm targets in Section 3.1.

race detection. 2D-Order provides a strong correctness guarantee — it reports a race if and only if the program has a race on that input.

Like prior work, 2D-Order has two main components: a reachability data structure and an access history component. Particularly, we find that 2D-dags have order-dimension two as SP-dags. Similar to SP-Order [9] and WSP-Order [97], as it executes the computation, 2D-Order maintains two total orders of the strands it has encountered. In Section 3.1, we define these two specific orders, show how to maintain them on the fly, and prove that maintaining these two orders suffices to answer the reachability query.

For parallel race detecting fork-join programs, Mellor-Crummey [55] proves that it suffices to store two readers and a single writer per memory location in the access history. We extend this result and show that two readers and one writer suffice for pipeline programs as well, which directly follows from the fact that the reachability data structure can be maintained with two orders.

The algorithm presented in Section 3.1 assumes that when a strand u is executed, we know how many children u has. This assumption may not hold for platforms that generate the pipeline dynamically. In Section 3.2, we present a variant of 2D-Order which only assumes that a strand v knows its parents when it executes; this assumption is generally true since v can only execute after all its parents have executed. This variant has the same performance bound.

The algorithms described in Sections 3.1 and 3.2 are formulated in terms of traversing a 2D-dag as the computation unfolds. In Section 3.3, we show how one can apply 2D-Order to the language constructs provided by Cilk-P [48, 49]. Cilk-P is an extension to the Cilk language that supports linear pipelines with a provably efficient work-stealing scheduler. Cilk-P is an interesting case study. Unlike most other systems, Cilk-P supports "on-the-fly" pipeline parallelism, where the structure of the pipeline emerges as the program executes.

We have also implemented PRacer, a prototype implementation of 2D-Order race detection algorithm applied to Cilk-P. Section 3.4 empirically evaluates the overhead of PRacer and shows that it incurs virtually no overhead for reachability maintenance and achieves similar scalability compared to applications' baseline executions without race detection.

3.1 2D-Order Algorithm

We now describe the basic 2D-Order algorithm. In this section, we make two simplifying assumptions: (1) We assume that a node u's children are known as soon as u finishes executing; and (2) There are no redundant edges — and edge from (u, v) is removed if there is already (a different) directed path from u to v. We will remove these assumptions in the next sections. We first provide some basic definitions before describing the 2D-Order's algorithm for reachability maintenance and proving its correctness. Then, we describe what information is kept in the access history and how 2D-Order checks for races. Finally, we prove the performance bound.

3.1.1 Notations and Definitions

Definition 1. *A **a two-dimensional dag (2D-dag)** is a planar directed acyclic graph with the following properties:*

1. *It has a unique source node s with no incoming edges and a unique sink node t with no outgoing edges.*

2. *Each node has at most two incoming and at most two outgoing edges. Edges are labeled as pointing either rightwards or downwards.*

This definition implies that each node can have at most two children — the down child of a node v is denoted by $v.dchild$ and the right child is denoted by $v.rchild$. Similarly, the up parent of v is denoted by $v.uparent$ and the left parent is denoted by $v.lparent$.

Definition 2. *Given two distinct nodes x and y, a node v is their* ***common ancestor*** *if $v \preceq x$ and $v \preceq y$. A node z is their* ***least common ancestor****, denoted by $lca(x, y)$, if for all common ancestors v of x and y, we have $v \preceq z$.*

By definition of a 2D-dag, a unique least common ancestor exists for any two nodes (proven in Lemma 9). The following lemma states that if $x \parallel y$, then their *lca* has two children, and x follows from one while y follows from the other.

Lemma 3. *For two nodes x and y, say $x \parallel y$ and $z = lca(x, y)$. Then we have (1) z has two children; (2) if $z.dchild \preceq x$ then $z.dchild \parallel y$, $z.rchild \preceq y$, and $z.rchild \parallel x$.*

Proof. Suppose that, for contradiction, $w \preceq x$ and $w \preceq y$ where w is a child of z; by Definition 2 $z \neq lca(x, y)$. □

This lemma allows us to relate any two parallel nodes.

Definition 4. *Given two nodes x and y where $x \parallel y$. Let $z = lca(x, y)$. Then, x is* ***down of*** *($\parallel_D$) y iff $z.dchild \preceq x$ & $z.rchild \preceq y$, and x is* ***right of*** *y ($\parallel_R$) iff $z.dchild \preceq y$ & $z.rchild \preceq x$.*

We now make some straightforward structural observations. (1) For distinct nodes x and y, exactly one of the following four conditions hold: $x \prec y$, $y \prec x$, $x \parallel_D y$ or $y \parallel_D x$. (2) Given a node x with two children, $x.dchild \parallel_D x.rchild$.

3.1.2 Reachability in 2D-Order Algorithm

2D-Order maintains two total orders on all strands using order-maintenance data structures. An ***order-maintenance (OM)*** data structure D maintains a total order of elements and provides the following operations.

- `OM-Precedes`(D, x, y): Given pointers to x and y, return *true* iff x precedes y in the total order kept by D.

- `OM-Insert`(D, x, y): Given a pointer to an existing element x, splice-in a new element y *immediately* after x in the total order. Thus, x and all its predecessors of x are before y in the total order, while all successors of x are after y.

Algorithm 1: 2D-Order: Reachability Maintenance

```
1  Function Insert-Down-First(v)
2      if v.rchild exists then
3          if v.rchild.uparent not exists then
4              OM-Insert (OM-DownFirst, v, v.rchild)
5      if v.dchild exists then
6          OM-Insert (OM-DownFirst, v, v.dchild)
7  Function Insert-Right-First(v)
8      if v.dchild exists then
9          if v.dchild.lparent not exists then
10             OM-Insert (OM-RightFirst, v, v.dchild)
11     if v.rchild exists then
12         OM-Insert (OM-RightFirst, v, v.rchild)
```

2D-Order keeps two OM data structures — called OM-RightFirst and OM-DownFirst — to maintain two different orders on all the nodes in the 2D-dag. The OM data structures for both orders are initialized by inserting the source node s as the first node. Subsequently, it executes nodes of the dag in any valid serial or parallel order — that is, a node can be executed when it's predecessors have finished executing. After executing each node v, 2D-Order calls the two functions shown in Algorithm 1.

Function `Insert-Down-First(`v`)` inserts v's children into the OM-DownFirst data structure. Immediately after this function is executed, the following will be true in the OM-DownFirst order: (a) If v has a down child v_D, then v_D will be immediately after v. (2) If v has a right child v_R and v_R doesn't have an up parent, then v_R will be immediately after v_D (if v_D doesn't exist, then v_R will be immediately after v). The symmetric invariant is true for the function `Insert-Right-First(`v`)`.

In other words, for any node u, its up parent is "responsible" for inserting it into the OM-DownFirst data structure and its left parent is "responsible" for inserting it into the OM-RightFirst data structure. If u doesn't have one of the parents, however, then u's other parent takes over the corresponding responsibility and inserts u immediately after its other child (or after the parent itself if the other child doesn't exist). It should be clear that each node u is inserted into each OM data structure exactly once and these insertions happen before u itself is executed.

To simplify notation, we say that $x \rightarrow_D y$ if x occurs before y in the OM-DownFirst data structure (that is, if `OM-Precedes`(OM-DownFirst, x, y) returns true). Similarly, we say $x \rightarrow_R y$ if `OM-Precedes`(OM-RightFirst, x, y) returns true. Note that since the algorithm never swaps the order of the nodes once they are inserted, the answer returned will be consistent once both x and y are inserted.

3.1.3 OM-DownFirst and OM-RightFirst Maintain Reachability Relationships

We will now prove that these two total orders are sufficient to fully specify the partial order of the dag. The following theorem, which we prove in the remaining subsection, shows that given any two nodes x and y, we can determine the relationship between them just by looking at the total orders maintained by OM-DownFirst and OM-RightFirst; if x is before y in both orders, then $x \prec y$; if y is before x in both orders, then $y \prec x$; otherwise $x \parallel y$.

Theorem 5. *Given nodes x and y in OM-DownFirst and OM-RightFirst, $x \prec y$ iff $x \rightarrow_D y$ and $x \rightarrow_R y$.*

We first prove some structural properties of 2D-dags. Lemma 6 (stated without proof) says that any subdag G' of a 2D-dag G that has a single source and a single sink is also a 2D-dag. Then, we prove that any source to sink path cuts 2D-dags into disjoint graphs with certain properties (Lemmas 7 and 8).

Lemma 6. *Consider two nodes a and b in a 2D-dag G where $a \prec b$. Construct a subdag G' such that $x \in G'$ if $a \preceq x \preceq b$ and edge $(x, y) \in G'$ if $(x, y) \in G$. Then G' is a 2D-dag with source a and sink b.*

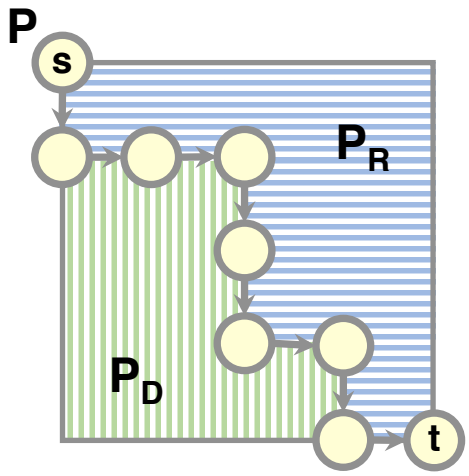


Figure 3.1: A path P divides a 2D-dag into two regions, P_R (shaded with horizontal lines) and P_D (shaded in vertical lines).

Consider a path P from the source s of a 2D-dag to its sink t. We use this path to divide all nodes of the dag into three subsets P, P_R and P_D. P contains all the nodes on the path. Note that for all $u \in P$, at least one of u's children (unless $u = t$) and one of u's parents (unless $u = s$) is also in P. Consider a node $u \notin P$ — since $s \in P$, u has at least one ancestor in P. Say q is the ancestor of u which is topologically latest in the path P. Since nodes on P are totally ordered, there is necessarily this latest node. Then, $u \in P_D$ if it follows from q's down child — that is, if $q.dchild \preceq u$. Similarly, $u \in P_R$ if $q.rchild \preceq u$. The intuitive meaning of P_R and P_D is shown in Figure 3.1 where path P cleanly divides the graph into two "contiguous regions." We now prove that the definition matches this intuitive meaning.

Lemma 7. *For any path P, $P_R \cap P_D = \emptyset$ and any path from node u to node v where $u \in P_R$ and $v \in P_D$ must include some node on P.*

Proof. First, we prove disjointness. Consider $u \notin P$ and say q is u's ancestor which is topologically latest in P; since at least one of q's children must be in P, u cannot follow from both its children. Now, assume for contradiction that there is a path from u to v that has no node in P. Then the latest ancestor of u and v on P must be the same node — which is a contradiction since $u \in P_R$ and $v \in P_D$. □

Lemma 8. *For any node $u \in P$,*

1. *If u has two children and $u.rchild \in P$, then $u.dchild \in P_D$. (Similarly, if $u.dchild \in P$, then $u.rchild \in P_R$.)*

2. *If u has two parents, if $u.lparent \in P$, then $u.uparent \in P_R$. (Similarly, if $u.uparent \in P$, then $u.lparent \in P_D$.)*

Proof Sketch. The first statement is obvious from definition, because $u \in P$ and u must be the latest ancestor of $u.dchild$ in P. The easy way to see the second statement is to notice that if we flip the direction of all edges and rotate the dag, then source becomes sink, $u.uparent$ becomes $u.dchild$, and $u.lparent$ becomes $u.rchild$, and P_R becomes P_D. To formally prove it, one can induct on the nodes on the path. □

We can now use Lemma 7 to prove that any two nodes have a unique lca.

Lemma 9. *Given two nodes x and y which are in parallel, $lca(x, y)$ exists uniquely.*

Proof. For the purpose of contradiction, assume two lcas exist, named z and z'. By the fact that they are both $lca(x, y)$, they must be in parallel with each other. Wlog, assume $z' \parallel_D z$ and let $u = lca(z', z)$. Also wlog assume that $x \parallel_D y$ with respect to z (i.e., x follows from $z.dchild$). Construct a path P that goes from the source s to u to $u.rchild$ to z to $z.dchild$ to x to the sink t; such a path exists by the property of 2D-dags and lcas. Then, as shown in Figure 3.2, $z' \in P_D$ (since it follows from $u.dchild$) and $y \in P_R$ (since it follows from $z.rchild$). However, since $z' = lca(x, y)$, there must be a path P' from z' to y, which must

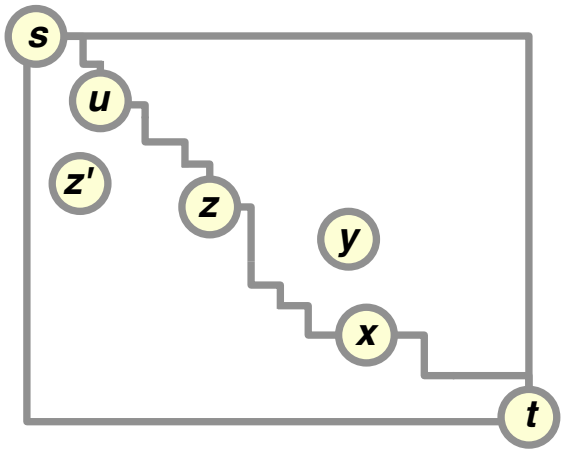


Figure 3.2: Figure for Lemma 9. Assume two lcas for x and y exist, namely z and z', and let u be a lca of z and z'.

cross P by Lemma 7. If P' cross P before z, it contradicts with the fact that no path exists between z and z'. If P' cross P after z, it contradicts with the fact that $z = lca(x, y)$. Thus, the $lca(x, y)$ must be unique. □

We now use these paths to prove a lemma about the insertion order between nodes.

Lemma 10. *At any point during the execution of 2D-Order, given node x which has two children. If $x.dchild \notin$ OM-DownFirst, then for any y such that $x.dchild \prec y$ and $x.rchild \parallel y$, $y \notin$ OM-DownFirst.*

Proof. Consider the 2D-subdag G' with source s and sink y. Let S be the set of all the nodes on paths from s to y. Now we construct path P using nodes from set S: for any node u such that $u \in S$ and $u \notin P$, we have $u \in P_D$. Intuitively, P follows the "top-most right-most" path among all paths from s to y.

First we prove $x \in P$. Suppose $x \notin P$, then by the constraint of path P, we have $x \in P_D$. Since $x.rchild \notin P$ because $x.rchild \parallel y$, we also have $x.rchild \in P_D$. Now construct a path P' from s to x to $x.dchild$ to y. By Lemma 8, $x.rchild \in P'_R$. Let $R = P'_R \bigcap P_D$ (the shaded region in Figure 3.3). Now consider the whole 2D-dag G with source s and sink t, which

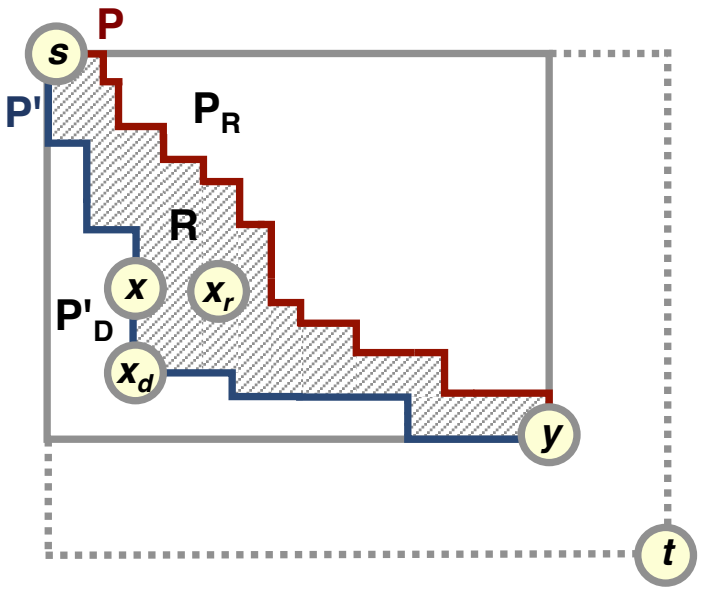


Figure 3.3: Figure for Lemma 10. Path P is shown in red, and path P' is in blue. The shaded region is R. The node x_d is $x.dchild$ and the node x_r is $x.rchild$.

includes R. Since $x.rchild \in R$ (the shaded region) and $t \notin R$, a path from $x.rchild$ to t must cross with either path P P'. Since both P and P' ends at y, this means $x.rchild \prec y$, which contradicts with the original assumption that $x.rchild \parallel y$. Therefore, we are guaranteed that $x \in P$.

Since P is a continuous path and $x.rchild \notin P$, we must have $x.dchild \in P$. Now we show P is actually an insertion chain, which means for any node w in P except for s, w is inserted by its parent in P.

Let v and w be two consecutive nodes in P and $v \prec w$. For the purpose of contradiction, let's assume that w is not inserted by v. According to the Down-First part of algorithm 1, this can only happen when w has two parents in G and $v = w.lparent$. Since $v \in P$, we have $v.uparent \in P_R$ by Lemma 8. However, since $v.uparent \in S$, it is guaranteed that either $v.uparent \in P$ or $v.uparent \in P_D$, which leads to a contradiction. Thus, P is an insertion chain. Since we know that $x.dchild \in P, y \in P$, and $x.dchild \prec y$, y cannot be inserted into OM-DownFirst before $x.dchild$. □

We can now prove the two important properties of down-first order — namely that $x \rightarrow_D y$ if $x \parallel_D y$ or if $x \prec y$.

Lemma 11. *At any point during the execution of 2D-Order, given nodes x and y in OM-DownFirst. If $x \parallel_D y$, then $x \rightarrow_D y$.*

Proof. We prove this lemma by induction. Suppose lemma is true before insertion is invoked on a node y. Consider any x in OM-DownFirst. We will show if $x \parallel_D y$, then $x \rightarrow_D y$. Let $z = lca(x, y)$; we have $z.dchild \preceq x$ and $z.rchild \preceq y$ (The case where $y \parallel_D x$ is similar.)

1. y **has one single parent** w: We first consider the case where $z = w$ and $z.rchild = y$. Since $z.dchild$ has not been inserted in OM-DownFirst yet, and since x must follow from $z.dchild$, from Lemma 10, we know x has not been inserted. So this case is trivial. Say $z.rchild \neq y$; then, we have $z = lca(w, x)$ and $x \parallel_D w$. By inductive hypothesis, we know $x \rightarrow_D w$. Because y is inserted immediately after w, therefore it's guaranteed that $x \rightarrow_D y$.

2. y **has two parents**: According to algorithm 1, y is inserted immediately after its up parent $y.uparent$, say w. We now argue that $x \parallel_D w$. Let $z' = lca(w, x)$. We know that $z' \preceq z$. If $z' = z$, then we are done. For the rest of the proof, we assume that $z \parallel w$; therefore, $z' = lca(z, w)$ since z' is an ancestor of z.

Assume for contradiction that $w \parallel_D x$; therefore $z'.dchild \preceq w$. Therefore, $z'.rchild \preceq z$. Consider the 2D-dag G' with source z' and sink t and consider a path P that goes from z' to $z'.dchild$ to w to y to t. We know that $z'.rchild \in P_R$ (from Lemma 8); therefore, $z \in P_R$ since otherwise, the path from z' to z will cross P at some node a and this node $a = lca(z, w)$ instead of z'. However, we also know that $y.lparent \in P_D$ (from Lemma 8) and $z \preceq y.lparent$. Therefore, the path from z to $y.lparent$ must cross P at some node b and this would mean that z is an ancestor of w which contradicts the assumption.

Therefore $x \parallel_D w$ and by IH, $x \rightarrow_D w$; therefore, when y is inserted immediately after w, we have $x \rightarrow_D y$. □

Lemma 12. *At any point during the execution of 2D-Order, given nodes x and y in OM-DownFirst. If $x \prec y$, then $x \rightarrow_D y$.*

Proof. We prove this lemma by induction. Suppose lemma is true before insertion is invoked on a node y. Consider any node x in OM-DownFirst.

We first show if $y \prec x$ then $y \rightarrow_D x$. Let w be the parent that inserted y. Since $y \prec x$, $w \prec x$. By IH, $w \rightarrow_D x$ before the insertion, and thus $w \rightarrow_D y \rightarrow_D x$ after the insertion.

Now we show if $x \prec y$, then $x \rightarrow_D y$.

1. y **has one single parent** w: In this case, y is inserted by this parent w immediately after w. Therefore, clearly, $w \rightarrow_D y$. Also, if $x \prec y$, then $x \preceq w$ since w is y's only parent. If $x = w$, we are done. If not, according to the inductive hypothesis, we have $x \rightarrow_D w$. Therefore, we know $x \rightarrow_D y$.

2. y **has two parents**: We know y is inserted immediately after $y.uparent$. If $x \preceq y.uparent$, we can apply the same argument as in the single parent case. Now consider the case that $x \parallel y.uparent$ and $x \preceq y.lparent$. We will show that $x \parallel_D y.uparent$ and by Lemma 11, we then have $x \rightarrow_D y.uparent$, which leads to $x \rightarrow_D y$.

For the purpose of contradiction, let's assume that $y.uparent \parallel_D x$ and let $z = lca(x, y.uparent)$. Consider the subdag with source z and sink t and construct a path P from z to $z.rchild$ to x to y to t. Then by Lemma 8, we have $y.uparent \in P_R$ and $z.dchild \in P_D$. Since $y.uparent \parallel_D x$, there must be a path P' from $z.dchild$ to $y.uparent$, which must cross P by Lemma 7, which contradicts the assumption that $z = lca(x, y.uparent)$. Thus we must have $x \parallel_D y.uparent$ and thus $x \rightarrow_D y$. □

Symmetrically, one can prove the following lemmas:

Lemma 13. *At any point during the execution of 2D-Order, given nodes x and y in OM-RightFirst. If $x \prec y$, then $x \rightarrow_R y$.*

Lemma 14. *At any point during the execution of 2D-Order, given nodes x and y in OM-RightFirst. If $x \parallel_R y$, then $x \rightarrow_R y$.*

Now we can prove the main result:

Proof of Theorem 5. From Lemma 12 and Lemma 13, it's straightforward to see that if $x \prec y$, then $x \rightarrow_D y$ and $x \rightarrow_R y$. For the other direction, suppose $x \parallel y$ when $x \rightarrow_D y$ and $x \rightarrow_R y$. Wlog, say $y \parallel_D x$. Then we have $y \rightarrow_D x$ by Lemma 11, which contradicts to $x \rightarrow_D y$. □

3.1.4 Checking Races and Updating Access History

Algorithm 2: 2D-Order: Access Histories

```
   /* Called when a strand r read memory location l                    */
 1 Function Read(r, l)
 2     if Precedes (writer(l), r) is false then
 3         ReportRace ( );
 4     if OM-Precedes (OM-RightFirst, dreader(l), r) then
 5         dreader(l) = r;
 6     if OM-Precedes (OM-DownFirst, rreader(l), r) then
 7         rreader(l) = r;
   /* Called when a strand w wrote to memory location l                */
 8 Function Write(w, l)
 9     if Precedes (writer(l).w) is false
10     or Precedes (dreader(l), w) is false
11     or Precedes (rreader(l),w) is false then
12         ReportRace ( );
13     writer(l) = w;
   /* When called, we have either u ≺ v or u ∥ v, but never v ≺ u      */
14 Function Precedes(u, v)
15     if OM-Precedes (OM-DownFirst, u, v)
16     and OM-Precedes (OM-RightFirst, u, v) then
17         return true
18     return false
```

We now describe how we do race detection using this algorithm — the code is shown in Algorithm 2. For each memory location ℓ, our algorithm stores at most one previous writer node — called ***last writer*** $writer(\ell)$ — this is simply the last node that wrote to

this memory location. If a set of reader nodes R_ℓ have read this memory, the algorithm stores up to two reader nodes: (1) ***downmost reader*** $dreader(\ell)$: For all $r \in R_\ell$, either $r \preceq dreader(\ell)$ or $r \parallel_R dreader(\ell)$; and (2) ***rightmost reader*** $rreader(\ell)$: For all $r \in R_\ell$, either $r \prec rreader(\ell)$ or $r \parallel_D rreader(\ell)$.

When a node u tries to write location ℓ, it uses the OM-DownFirst and OM-RightFirst data structures to check whether either $dreader(\ell) \parallel u$, $rreader(\ell) \parallel u$ or $writer(\ell) \parallel u$. If so, we report a race. In either case, u is now last writer $writer(\ell)$. When node u tries to read location ℓ, it uses the OM-DownFirst and OM-RightFirst data structures to check whether $writer(\ell) \parallel u$. If so, we report a race. In either case, u now checks if $dreader(\ell) \rightarrow_R u$; if so, u is the new downmost reader $dreader(\ell)$. Similarly, if $rreader(\ell) \rightarrow_D u$, then u is the new $rreader(\ell)$.

Theorem 15. *2D-Order never reports false races and for racy programs, reports at least one race.*

This is the standard correctness guarantee for on-the-fly race detection algorithms. The proof mostly follows from previous results — with the exception of one wrinkle. It is always sufficient to store a single writer in access history; however, it is not always sufficient to store two readers. In particular, for general dags, one has to store all parallel reads that happened since the last write. Mellor-Crummey [55] proved that for series-parallel dags, it is sufficient to record two readers. We will now show that for 2D-dags, it is also sufficient to store two readers — in particular, the downmost and the rightmost readers.

First, let us notice that at any point in the execution $rreader(\ell)$ (and $dreader(\ell)$) is unique (or null if no node has read ℓ yet). To see this, note that from Algorithm 2, lines 6 and 7, $rreader(\ell)$ is simply the last node in the order maintained by OM-DownFirst that read ℓ.

Theorem 16. *At any point during the execution of a 2D-dag, let R_ℓ be the set of nodes that have read memory location ℓ and w be any other node. We have $r \prec w$ for all $r \in R_\ell$ if and only if $dreader(\ell) \prec w$ & $rreader(\ell) \prec w$.*

Proof. It is clear that if all $r \in R_\ell$ precede w then so do $dreader(\ell)$ and $rreader(\ell)$ since they belong to the set R_ℓ.

Now say that $dreader(\ell) \prec w$ & $rreader(\ell) \prec w$. For any $r \in R_\ell$, we have, $r \rightarrow_R dreader(\ell) \rightarrow_R w$ (the first arrow is by definition of downmost reader and the second from Theorem 5). Similarly, we have $r \rightarrow_D rreader(\ell) \rightarrow_D w$. Therefore r is before w in both OM-DownFirst and OM-RightFirst orders; by Theorem 5, we know that $r \prec w$. □

3.1.5 Performance of 2D-Order

Theorem 17. *For a 2D-dag G with work T_1 and span T_∞, we can run do race detection using 2D-Order in time $T_1/P + T_\infty$ time on P processors.*

First, each node is inserted at most once in OM data structures and every memory access requires a constant number queries to OM data structures. If inserts and queries to OM data structures took $O(1)$ time, then the work of the program augmented with 2D-Order is $O(T_1)$ and the span is $O(T_\infty)$.

Sequentially, an OM-data structure can be implemented for $O(1)$ cost per operation (amortized) [8, 25]. This immediately gives us an $O(T_1)$ time (optimal) sequential algorithm. This slightly improves the best previous result [26], which has a multiplicative overhead of the inverse Ackermann's function (which is, admittedly, small in practice).

To get parallel performance, we need an OM implementation that supports concurrent operations. No general $O(1)$-time-per-operation concurrent OM data structure is known. Utterback et al. [97] provide an algorithm (containing modified OM data structure and work-stealing scheduler) for programs that access OM data structures in a ***conflict-free*** way. In particular, if a parallel program guarantees that two logically parallel strands will

never try to insert immediately after the same node, then they show the following result (it is not explicitly stated, but is implied from their proofs).

Lemma 18. ***From [97]*** *A parallel program with work T_1 and span T_∞ which accesses into OM data structure(s) in a conflict-free way can be executed in time $O(T_1/P + T_\infty)$ time.*

Note that 2D-Order follows the conflict-free restriction since all inserts after node v occur when v executes. Therefore, Theorem 17 follows directly from this lemma. Note that this performance bound only holds if we use the particular implementations of both the work-stealing scheduler and the OM data structure described in Utterback [97]. In Section 3.4, we briefly describe how we adapted this scheduler for Cilk-P runtime system.

3.2 Generalizing 2D-Order

In Section 3.1, we made two assumptions: (1) When we execute a node, we already know both its children and whether these children have their other parent. In practice, we may not have this information until we encounter the child node. (2) There are no redundant edges.

Algorithm 3 shows a variant of 2D-Order — these functions are called immediately before executing node v. Here, when v is executed, instead of inserting its real children, 2D-Order creates two placeholder nodes for both of v's children (denoted as $dchild_h$ and $rchild_h$). It will insert both nodes into OM-DownFirst and OM-RightFirst orders. As seen on lines 6, 7, 13, and 14 the order after all insertions is $v \rightarrow_D v.dchild_h \rightarrow_D v.rchild_h$ and $v \rightarrow_R v.rchild_h \rightarrow_R v.dchild_h$. This is consistent with Algorithm 1 except here we assume that both children exist and always insert them regardless of the presence of the other parent.

When a node is executed, it finds its corresponding placeholder nodes by accessing its parents. If v has only one parent, it has only one placeholder node in each data structure, and 2D-Order simply use this placeholder node to represent v in the future. Now consider a node v that has two parents. Both parents will insert a placeholder node to represent v in

OM-DownFirst and OM-RightFirst data structures, possibly at different positions. When v is executed, 2D-Order chooses one of these dummies as the "real" one (and different ones in OM-DownFirst and OM-RightFirst) which will be used henceforth to represent v when accessing each order. In particular, whenever we access OM-DownFirst, the placeholder inserted by v's up parent will represent v.[4] Correspondingly, when we access OM-RightFirst, the placeholder inserted by v's left parent will represent v. The access history and queries are not affected.

Algorithm 3: Variant 2D-Order

```
1  Function Insert-Down-First(v)
2      if v.uparent exists then
3          dCurr = v.uparent.dchild_h;
4      else
5          dCurr = v.lparent.rchild_h;
6      OM-Insert (OM-DownFirst, dCurr, v.rchild_h);
7      OM-Insert (OM-DownFirst, dCurr, v.dchild_h);
8  Function Insert-Right-First(v)
9      if v.lparent exists then
10         rCurr = v.lparent.rchild_h;
11     else
12         rCurr = v.uparent.dchild_h;
13     OM-Insert (OM-RightFirst, rCurr, v.dchild_h);
14     OM-Insert (OM-RightFirst, rCurr, v.rchild_h);
```

The code for handling redundant edges is not shown, but is straightforward. When a node x has two parents, it first checks if either of them precede the other (using OM-DownFirst and OM-RightFirst) and if so, it ignores the redundant edge.

Lemma 19. *Algorithm 3 has the same correctness and performance properties as Algorithm 1.*

[4]The placeholder inserted by the v's left parent will never be accessed in OM-DownFirst if v also has an up parent — this node becomes a dummy. As an optimization, we can remove this node from OM-DownFirst; however, this has no bearing on the theoretical correctness or performance.

The intuition behind the omitted proof is that Algorithm 3 maintains the same order as Algorithm 1 — node v finds its correct representatives right before it is executed. Before v executes, the placeholder nodes for v will never be used in any queries or to insert any other nodes. For the performance guarantee, notice that each function call does at most twice as many inserts as Algorithm 1.

3.3 PRacer: Race Detection for Cilk-P

This section describes PRacer, the particular implementation of 2D-Order when applied to Cilk-P [48, 49]. Race-detection for Cilk-P is an interesting case study because Cilk-P's language constructs allow for much more dynamism in the structure of the pipeline. Due to the particular quirks to Cilk-P's pipelines, PRacer incurs an additional $\lg k$ overhead on the span term, where k is the vertical length of the 2D-dag. This section reviews Cilk-P's support for pipeline parallelism, presents PRacer in terms of Cilk-P's pipeline constructs, and explains the additional performance overhead.

3.3.1 Cilk-P's Support for Pipeline Parallelism

From a programmer's perspective, a linear pipeline is simply a loop over a stream of input elements, where each loop ***iteration*** i processes the ith element of the input stream. The loop body encodes the sequence of ***stages*** — abstract functions through which input elements are processed. These pipelines allow for parallel execution since the execution of iterations can overlap in time

Cilk-P extends Cilk with three keywords: `pipe_while`, `pipe_stage`, and `pipe_stage_wait`. The keyword `pipe_while` denotes a loop that can be executed in parallel in a pipelined fashion. The first stage of each iteration is stage 0 and `pipe_while` ensures that there are sequential dependences across stage 0 of all iterations; that is, stage 0 of iteration i does not begin until stage 0 of iteration i-1 completes. The keywords `pipe_stage`

and `pipe_stage_wait` are used inside the body of a `pipe_while` loop to denote stage boundaries. By default, the execution of these constructs in stage s of iteration i ends stage s and advances to stage s+1 of iteration i. Keyword `pipe_stage_wait` is used to enforce dependences between adjacent iterations. Ending a stage s of iteration i with `pipe_stage_wait` enforces that execution of stage s+1 of iteration i does not begin until stage s+1 of iteration i-1 finishes. Finally, `pipe_while` implicitly has a cleanup stage at the end of every iteration that occurs sequentially across iterations.

The structure of the pipeline for Cilk-P programs is determined dynamically at runtime. The `pipe_stage` and `pipe_stage_wait` statements can be enclosed within other control constructs, which allows programmers to dynamically vary the number of stages and enforce dependences based on the input to the iteration. Furthermore, `pipe_stage` and `pipe_stage_wait` optionally take a ***stage number***, an integer argument to name the stage that the statement is advancing to. This gives the programmer the flexibility to dynamically determine the label of stages and skip stages.

Cilk-P constructs can generate the example dag shown in Figure 3.4. Each iteration is a vertical line. There are sequential dependences across the first and last stages of all iterations, as dictated by `pipe_while`. The stages of an iteration form a chain, and horizontal dependences are enforced by `pipe_stage_wait`. By naming stages in a certain way, the programmer can skip stages or dictate that the stages to be labeled certain way. Still, Cilk-P's pipeline constructs always generate a dag that satisfies Definition 1 in Section 3.1.

Cilk-P incorporates a work-stealing scheduler that can schedule the resulting computation efficiently. Given a computation with T_1 work and T_∞ span, Cilk-P schedules the computation on P processors in expected time $T_1/P + O(T_\infty)$.[5]

[5]All bounds stated in this section are expected time bounds. Analogous high probability results can be obtained by applying standard techniques.

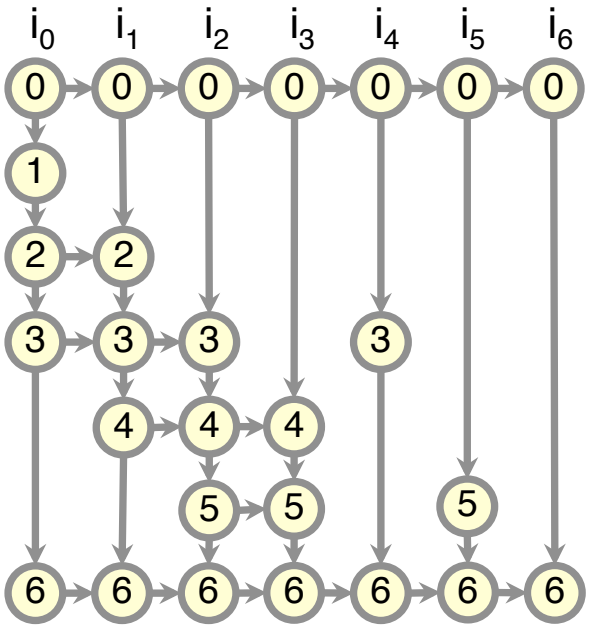


Figure 3.4: An example of the kind of 2D-dag Cilk-P can generate. A node presents a strand, and an edge denotes dependence between two strands. The iteration numbers are denoted above, and the numbers in the nodes denote the stage numbers.

3.3.2 PRacer: Applying 2D-Order to Cilk-P

Algorithm 4 shows the pseudocode for applying 2D-Order to Cilk-P's pipeline constructs. In Cilk-P, nodes do not know if they have a right child when they execute. Stage s of iteration i does not know if stage s of iteration i+1 will depend on it; we only find out that this dependence exists when (and if) stage s-1 of iteration i+1 calls `pipe_stage_wait`. Therefore, like in Algorithm 3, we must employ placeholder nodes. The function `StageFirst` is called before executing stage 0 of an iteration and is similar to Algorithm 3. The main difference is that, since stage 0 has no *uparent*, it knows to use the $rchild_h$ from its *lparent* (stage 0 of the previous iteration) as its representative. The function `StageNext` is called when `pipe_stage` is executed — again, a stage initiated by `pipe_stage` has no *lparent* and knows to use the $dchild_h$ from its *uparent* (the previous stage in the same iteration).

Algorithm 4: 2D-Order for Cilk-P

```
1  Function StageFirst(i)
2      if i is 0 then
3          dCurr = rCurr = source;
4      else
5          dCurr = rCurr = stage[i − 1][0].rchild_h;
6      InsertPlaceHolder (dCurr, rCurr, stage[i][0]);
7  Function StageNext(i, s)
8      dCurr = rCurr = stage[i][s − 1].dchild_h;
9      InsertPlaceHolder (dCurr, rCurr, stage[i][s]);
10 Function StageWait(i, s)
11     dCurr = stage[i][s − 1].dchild_h;
12     left = FindLeftParent (i, s);
13     if left ≠ −1 then
14         rCurr = stage[i − 1][left].rchild_h;
15     else
16         rCurr = stage[i][s − 1].dchild_h;
17     InsertPlaceHolder (dCurr, rCurr, stage[i][s]);
18 Function InsertPlaceHolder(dCurr, rCurr, stage)
19     OM-Insert (OM-DownFirst, dCurr, stage.rchild_h);
20     OM-Insert (OM-DownFirst, dCurr, stage.dchild_h);
21     OM-Insert (OM-RightFirst, rCurr, stage.dchild_h);
22     OM-Insert (OM-RightFirst, rCurr, stage.rchild_h);
```

The interesting function is `StageWait`, which is called when `pipe_stage_wait` executes and the execution is ready to advance to the next stage (i.e., dependence from the previous iteration has been satisfied). Since a stage initiated by `pipe_stage_wait` has both *uparent* and *lparent*, the order maintained by OM-DownFirst should use the $dchild_h$ from its *uparent*, and the order maintained by OM-RightFirst should use the $rchild_h$ from its *lparent*. However, since Cilk-P allows the execution to skip stages, identifying a stage's *lparent* requires additional work.

Consider the example shown in Figure 3.4. Say stage 5 of iteration i_5, denoted as (i_5, 5), had been created with `pipe_stage_wait(5)` instead of `pipe_stage(5)`. Since iteration i_4 does not have a stage 5, the left parent of (i_5, 5) is (i_4, 3). Consider another example case: Say stage (i_4, 3) had been created with `pipe_stage_wait(3)`. This would result a dependence from stage (i_3, 0) to stage (i_4, 3); however, this dependence is already subsumed by the dependences from (i_3, 0) to (i_4, 0) and from (i_4, 0) to (i_4, 3). In this case, (i_4, 3) does

not have a *lparent* despite being created by `pipe_stage_wait(3)`. The invariant is the following: When a stage (i, s) is initiated by `pipe_stage_wait` and stage (i-1, s) does not exist, (i, s)'s *lparent* is (i-1, s') where s' is the largest stage in iteration i-1 such that $s' < s$ and (i-1, s') is logically in parallel with the *uparent*, (i, s-1). Otherwise, (i, s) does not have an *lparent*.

Function `FindLeftParent`, called in Algorithm 4, line 12, performs the additional work needed to identify a stage's *lparent* (or lack thereof). The pseudocode for this function is not shown since it is a little complex. Briefly, for every active iteration i, we keep some metadata for the previous iteration i-1; in particular, we keep an in-order array of the stage numbers i-1 has executed so far. When `FindLeftParent` is called in for stage (i, s), we search this array to find the correct *lparent*, and `-1` is returned if *lparent* does not exist.

Execution Time. The only additional work in Algorithm 4 (compared to Algorithm 3) is `FindLeftParent` — this function must be carefully implemented to minimize overhead. Consider the obvious option. We can do a binary search on the metadata array — since `FindLeftParent` could be called for every node and there can be T_1 nodes, this can add a $\lg k$ multiplicative overhead, leading to a time bound of $O(\lg kT_1/P + \lg kT_\infty)$, where k is the maximum array size.

Observe that if `FindLeftParent` is called by different stages of the same iteration, the answers returned are strictly increasing — if *lparent* of stage s is s', no subsequent stage can have an *lparent* smaller than s'. Thus, within `FindLeftParent`(i, s) we can search the metadata array of iteration i-1 linearly starting from the smallest stage, removing all stages smaller than s' from the array. Each item is removed at most once and the cost of the search is at most the number of items removed, allowing us to amortize the work of searches against the work of the nodes removed. However, it has the disadvantage that some calls to `FindLeftParent` may cost up to k. All expensive searches may happen on the span, giving us the worst case bound of $O(T_1/P + kT_\infty)$.

`FindLeftParent`(i, s) implements a strategy that combine the best of both worlds. Say the previous iteration (i-1) has k elements in its metadata array. We start from the smallest and look at $\lg k$ elements linearly. If we find our *lparent*, we remove all elements smaller than s and return. If not, we can remove all $\lg k$ elements we looked at, since they are clearly smaller than s. Next, we do a binary search on the rest of the metadata array to find the correct answer. Note that the cost of each search is $O(\lg k)$. In addition, if the cost was c, we removed $\Omega(c)$ elements from the metadata array. Therefore, we can amortize the work in the same manner and only incur a $\lg k$ overhead on the span, giving us the bound of $O(T_1/P + \lg k \cdot T_\infty)$ for PRacer.

Composability with Fork-Join Parallelism. Cilk-P allows programmers to compose fork-join and pipeline constructs. Each stage can itself be an SP-dag or a 2D-dag and the nesting can be arbitrarily deep. Since nested 2D-dags are also 2D-dags, PRacer obviously applies directly when pipelines are nested inside pipelines. We now describe how we can handle nested fork-join parallelism.

2D-Order's reachability maintenance algorithm is similar in spirit to WSP-Order [97]. Recall from Section 2.4, WSP-Order keeps track of two total orders of the executed strands: English order and Hebrew order. The two strands are logically in parallel if and only if their relative order in English and Hebrew differ. English order is analogous to OM-DownFirst and Hebrew order is analogous to OM-RightFirst.

Nested fork-join parallelism is handled in a straightforward manner. When a stage is an SP-dag, we simply insert the nodes of this dag in English order in OM-DownFirst structure and in Hebrew order in OM-RightFirst structure. Reachability relationships are still checked by comparing relative orders of strands in the two structures. It is also straightforward to see why this is correct. Imagine a stage that was a single node u, represented by a single element in the OM data structures. When this node is replaced by an SP-dag G', this algorithm will replace the representative element of u in OM-DownFirst by the all the nodes in G in English

order and in OM-RightFirst by nodes in G in Hebrew order. All nodes in G would have the same relationship with other nodes in the pipeline as u did.

3.4 Performance Evaluation

This section summarizes the implementation of PRacer and evaluates its practical performance on three benchmarks in terms of overhead and scalability. We first evaluate the performance of only the reachability maintenance — that is, each node is inserted into OM data structures as shown in Algorithm 4, but memory accesses are not instrumented. These experiments indicate that the overhead of 2D-Order's reachability maintenance less than 1% in all benchmarks we examined, and it provides similar scalability as the baseline program without reachability maintenance. We also evaluated the full race detection algorithm including access histories. In this case, as with all race detection algorithms, the overhead is significant, between 14.7–41.1× overhead compared to the baseline. However, we still get scalability similar to the baseline; therefore, some of the overhead can be offset by running race detection in parallel.

Implementation of PRacer. We implemented PRacer by extending an open-source Cilk-P runtime system, released by Intel [45], which supports the pipeline constructs as macro-defines and the corresponding work-stealing scheduler to schedule them. The implementation of PRacer consists of two components, the race detection tool component and the runtime component.

The tool component ensures that functions shown in Algorithm 4 are called at the appropriate time to perform insertions into the two OM data structures, queries are performed on memory accesses, and manages metadata for `FindLeftParent` and access histories. The tool component is called via instrumentation inserted into Cilk-P control constructs (described in Section 3.3.1) and memory references. We enabled the instrumentation of pipeline constructs by modifying the macros defining the pipeline constructs in Cilk-P. For

memory accesses, we piggyback on the ThreadSanitizer instrumentation [84] that came with the LLVM/Clang compiler (version 3.4.1).

The runtime component required significant modification to the Cilk-P's work-stealing scheduler to allow for concurrent OM data structures based on the scheme described by Utterback et al. [97]. At a high-level, their concurrent OM data structure does ***parallel rebalances*** — occasionally, large portions of the data structure may be re-organized in parallel. The work-stealing scheduler must be designed to (1) perform appropriate concurrency control so that inserts do not occur during a parallel rebalance; and (2) appropriately move workers between the main program and the parallel rebalance. Utterback et al. implemented their system in open-source Cilk Plus runtime system released by Intel [42]. For PRacer, we re-implemented their strategy in the Cilk-P runtime system since the original runtime does not support pipelines.

Experimental Setup. We use three benchmarks to evaluate PRacer: `ferret`, `lz77`, and `x264`. Benchmark `ferret` performs content-based similarity search on images. Benchmark `lz77` is a lossless, dictionary file compression algorithm. Benchmark `x264` is an video encoder. Both `ferret` and `x264` are from PARSEC benchmark suite [10] and modified to use Cilk-P's pipeline constructs. They are both evaluated using the largest input data set, `native`, that comes with PARSEC. We implemented `lz77` from scratch and ran it with an input text file of size 162-MBytes.

	stages / iter	*# of iters*	*# of reads*	*# writes*
`ferret`	5	3501	1.23e11	1.23e10
`lz7`	3	162	8.96e10	2.97e10
`x264`	71	36352	1.12e12	1.17e11

Table 3.1: The execution characteristics of the benchmarks.

Table 3.1 shows the characteristics of these benchmarks. Both `ferret` and `lz77` have relatively simple pipelines, where the structure of the pipeline is static and has a fixed

number of stages across iterations, five and three respectively. On the other hand, `x264` utilizes the on-the-fly feature of Cilk-P's pipeline parallelism — even though the number of stages across iterations are the same, they can take on different stage numbers from one iteration to another.

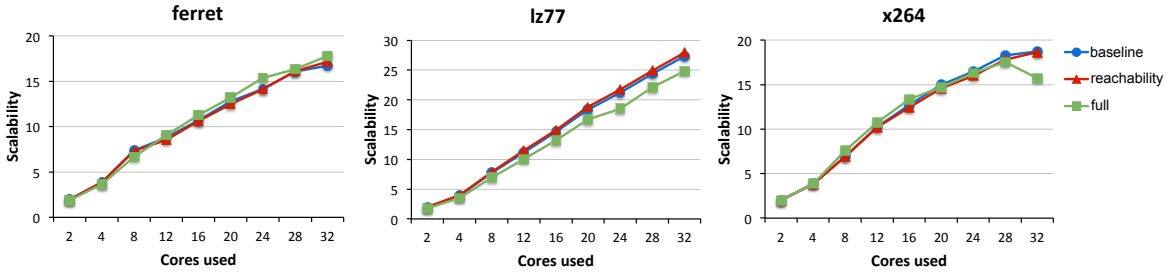


Figure 3.5: The scalability of the benchmarks. The x-axis shows the number of cores used. The y-axis shows the scalability, computed by taking the runtime on one core divided by the runtime on P cores under the same configuration, where P is the number cores used.

We ran all our experiments on an Intel Xeon E5-4620 with 32 2.20-GHz cores on four sockets. Each core has a 32-KByte L1 data cache, 32-KByte L1 instruction cache, a 256-KByte L2 cache. There are a total of 500 GByte of memory, and each socket share a 16-MByte L3-cache. All benchmarks are compiled with LLVM/Clang version 3.4.1 with `-O3` running on Linux kernel version 3.10. Each data point is the average of 10 runs with standard deviation less than 5%.

Overhead of PRacer. To get a sense of PRacer's overhead breakdown, we ran the benchmarks with three different configurations: the ***baseline*** configuration, which is the the original program without race detection, the ***reachability***, which is the execution with only the reachability component of the 2D-Order without memory instrumentation; and ***full***, which is the execution with the full 2D-Order including both the reachability maintenance and access history management.

	baseline	*reachability*	*full*
`ferret`	191.902	191.987 (1.00×)	7984.067 (41.60×)
`lz77`	116.079	117.902 (1.02×)	1703.636 (14.68×)
`x264`	933.721	934.572 (1.00×)	15877.110 (17.00×)

Table 3.2: The execution times for the benchmarks running on one core for all configurations, shown in seconds. The numbers in parentheses indicate the overhead compared to the baseline.

Table 3.2 shows the sequential (T_1) running time for all of the three configurations.[6] The overhead due to reachabilty maintenance is insignificant for all benchmarks. On the other hand, adding memory instrumentation increases overheads significantly. This is explained by the fact that each stage is inserted at most twice in each OM data structure and the number of stages is relatively small (2.5e6 for `x264`). The total number of memory accesses is many orders of magnitude larger. However, these results are consistent with the overhead of full race detection in the literature [97].

Scalability of PRacer. Finally, we show that PRacer scales similarly compared to the baseline. Figure 3.5 shows the scalability plot of the three benchmarks. As can be seen in the plots, the scalability of the reachability maintenance and the full configurations track closely to that of the baseline. This scalability is especially useful since race detection is so expensive — serially, `x264` takes 4 hours with full race detection. The parallelism of PRacer cuts this running time to a more reasonable amount for debugging.

[6]The running times for `x264` are much slower than what was shown in the literature, because we disabled the vectorization code in order to perform race detection correctly.

Chapter 4

Futures and Proactive Work-Stealing

The use of future constructs provides a flexible way to express parallelism. Similar to fork-join parallelism, one can spawn off a ***future task*** that executes logically in parallel with the continuation of the spawn statement. Unlike fork-join parallelism, however, the termination of a future task is not restricted to a lexical scope. Rather, the spawn statement returns a future handle that can be used to retrieve the value produced by the future task. When the handle is ***touched***, the control is blocked until the corresponding future task terminates and returns a value.

The additional flexibility of futures allows one to write a wider range of parallel programs and/or provide higher level of parallelism beyond what can be specified using only fork-join parallelism. For instance, Blelloch and Reid-Miller [11] show that, one can asymptotically reduce the span of various tree operations using parallel futures. Since its proposal in the late 70th [6, 33], the future constructs have been incorporated into various task parallel languages and platforms [16–18, 32, 36, 47, 51, 88, 94], including the C++11 standard [46].

However, such flexibility comes with a cost. Even though the classic work-stealing (previously described in Section 2.3) and its execution time bound apply to programs that use futures [4, 5], prior works [1, 37, 87] show that, when scheduled using classic work-stealing,

a program with futures, compared to a program that uses only fork-join parallelism, can incur much higher number of "deviations" — a better metric for evaluating the performance of parallel executions.

As articulated by Spoonhower et al. [87], the number of deviations provides a better metric for evaluating performance bounds because it is highly correlated to the additional cache misses and scheduling overheads of parallel executions. Informally, a ***deviation*** occurs during a parallel execution when a processor executes an instruction whose ordering in the instruction stream deviates from that of the serial execution. A deviation forces the scheduler to perform additional bookkeeping to keep track of the events that cause the deviations. Moreover, the number of deviations can be used to bound the extra cache misses incurred on the private caches during parallel executions (as first shown by Acar et al. [1]) — intuitively, the bound holds by considering each deviation to execute with an empty private cache.

Given a computation that employs only fork-join parallelism, Acar et al. [1] show that, the expected number of deviations[7] incurred by a classic work-stealing scheduler is $O(PT_\infty)$. In contrast, given a computation that employs k future operations, Spoonhower et al. [87] show that the expected number of deviation incurred is $O(PT_\infty + kT_\infty)$, an additional k multiplicative factor. More recently, Herlihy and Liu [37] show that, if futures are used in a *restricted* fashion, one can bound the number of deviations to be $O(PT_\infty^2)$.

All the prior works assume a ***parsimonious work-stealing*** scheduler, in which each worker maintains a single deque and only steals to load balance when its queue becomes empty. Due to the parsimonious nature of work-stealing analyzed, each future touch can lead to $O(T_\infty)$ number of deviations, contributing to the $O(T_\infty)$ multiplicative factors in the deviation bound.

To minimize the deviations caused by futures, therefore, we propose an alternative scheduling approach, called ***proactive work-stealing (ProWS)***: whenever a worker thread

[7] Acar et al. refer to deviations as drift nodes in work [1].

encounters a future touch that is not ready, it suspends the execution of its current task and tries to find something else to do. By proactively suspending the computation instead of expanding on what's already on the deque, one can minimize the deviations and thus the corresponding scheduling overhead and cache misses.

We show that ProWS can provide a comparable execution time bound to the parsimonious variant, as well as equal or better bounds on the number of deviations for programs that use futures. Given a computation that employs futures with T_1 work and T_∞ span, the proposed algorithm executes the computation on P processors in $O(T_1/P + T_\infty \lg P)$ time, which is asymptotically comparable to the parsimonious version (except for the $\lg P$ overhead on the span term). For ***structured use of futures***, where the future is single-touch with no races on the future handle, the algorithm incurs $O(PT_\infty^2)$ number of deviations, the same as the bound for the parsimonious variant. For ***general use of futures***, where the only restrictions are a constant number of touches per future and deadlock-freedom during one-worker execution,[8] the algorithm incurs $O(m_k T_\infty + PT_\infty \lg P)$ deviations, where m_k is the maximum number of future touches that are logically parallel. This bound is better than the bound for the parsimonious variant if $m_k = \Omega(P \lg P)$ and is smaller than k, the total number of touches in the entire computation; these assumptions hold true for all the benchmarks examined. Since proactive and parsimonious work-stealing behave the same for programs that utilize only fork-join parallelism, they have the same bounds for such programs.

We have implemented a work-stealing runtime system called Cilk-F, by extending Cilk Plus [44], a task parallel runtime system, to incorporate support for parallel futures scheduled using ProWS. We empirically evaluate Cilk-F and show that ProWS can be implemented efficiently.

[8]Prior work by Spoonhower et al. [87] assumes single-touch per future, but constant number of touches does not change their bound.

Contribution statement

The results of this chapter is from the joint work with Kyle Singer and I-Ting Angelina Lee. Specifically, the author of this dissertation is a major contributor to the following contributions:

- ProWS, a proactive work-stealing algorithm for scheduling computations with futures (Section 4.2).
- Theoretical proof that ProWS provides equal or better bounds on the number of deviations than the parsimonious variant (Section 4.3).

4.1 Future Parallelism

Like fork-join parallelism, future parallelism can also be expressed using two simple keywords: `create` and `get`. Similar to `spawn` in fork-join code, `create` can be used to create parallelism. A function F may spawn off a function G representing a ***future task*** by prefixing the call to G with `create`, and the continuation of F may execute in parallel with G. Unlike `spawn`, however, the termination of G is not confined to the enclosing lexical scope of the call, and the execution of a `sync` in F has no effect on it. Rather, the `create` call returns a ***future handle***, which can be used later to ensure termination of G and retrieve the result of its evaluation. One can invoke `get` on the future handle, an operation referred to as the ***future touch***, which causes the control to block until the corresponding future task terminates. Implicitly, we assume that the end of a future task always executes a `put`, depositing the task's resulting value into its handle.

4.1.1 Modeling Future Parallelism

Recall how Section 2.2 described modeling fork-join parallelism. Similar to the execution of a `spawn` in fork-join code, the `create` keyword terminates the current stand, which is a

create node with two outgoing edges: one to the first strand in the future task, and one to the continuation of `create`. A future touch, or invocation of `get`, terminates the currently executing strand and creates a ***join node*** that has two incoming edges: one from the strand that was terminated by the invocation of `get`, referred to as the ***local parent*** of the join node, and one from the last strand of the corresponding future task that executes the `put`, referred to as the ***put node*** and the join node's ***future parent***.

For ease of description, we refer to the edge that goes from a future create node to the first strand of the spawned future task as a ***create edge***. We will refer to the edge that goes from the last strand of a future task (put node) to the corresponding future join node as a ***join edge***.

When the program uses futures, the computation can be modeled as multiple independent SP-dags, connected via create edges and join edges. That is, if F spawns G via `spawn`, then the SP-dag of G is part of the SP-dag of G. On the other hand, if F spawns G via `create`, then F and G are independent SP-dags, with the first strand of G being the source of a separate SP-dag and the last strand of G being the sink.

As customary to prior works, we shall assume that the spawned function or future task is always the left child of the spawn node and the continuation strand the right child. Thus, a ***serial (one-worker) execution*** of a computation dag follows the left-to-right depth-first traversal. This also means that we assume eager evaluation of futures, where the future task is always evaluated before the continuation of `create` under serial execution.

Given a dag, the serial execution imposes a total order on the nodes. Say in this total order, v executes immediately after u. In a parallel execution, if a worker w executes v but not immediately after it executes u, then we say v incurs a ***deviation***. This could happen either because a different worker executed u or because worker w executed something else between u and v.

4.1.2 Types of Futures

A ***structured*** use of futures imposes the following restrictions: 1) single touch, meaning that only a single `get` is invoked on each future handle, and 2) no race on a future handle, meaning that there is a directed path between a future create node and the local parent of its corresponding touch. Note that this restriction is the same as prior work [37] and does not preclude a future task to execute in parallel with the function that performs its touch before the `get` keyword. It simply means that the spawning of the future (which writes to the future handle) must be in series with the invocation of the corresponding `get` (which reads the future handle). A ***general*** use of futures imposes the following restrictions: each future is touched a constant number of times and all the join edges are ***forward pointing***, namely, a `create` is always before its corresponding `get` in a serial execution, to prevent deadlocks.

4.2 Proactive Work-Stealing

This section describes the proactive work-stealing algorithm, which we shall refer to as ProWS in the rest of the section. We will refer to the original parsimonious algorithm (classic work-stealing) analyzed by Arora et al. [4] as ABP.

The main distinction between ProWS and ABP is as follows. When a worker executes a `get`, the associated future task may not be ready, so executing the `get` does not enable the subsequent future join node. With ProWS, this simply falls under the case of enabling zero nodes, and the worker continues execution by popping off the bottom-most node to execute next. ProWS handles the execution of `get` differently. If its future task is not ready, the worker ***suspends*** the entire deque and tries to find work elsewhere. An important consequence of such behavior is that there can be more than P deques in the system, where P is the number of workers.

In ProWS, suspended deques are still stored in a distributed fashion, thus each worker now manages a single ***active*** deque that it actively works on and a set of ***stealable*** deques that are not being actively worked on but contains ready nodes. When stealing, once a victim is chosen, a thief can steal from any deque that belongs to the victim with equal probability (including its active deque).

4.2.1 Data Structures Used

We shall first discuss the data structures used by the algorithm. Each deque supports the following operations:

- `popTop`: remove and return the node from the top;
- `popBottom`: remove and return a node from the bottom;
- `pushBottom`: insert a node onto the bottom;
- `pushBottomImplicit`: insert a node onto the bottom of the deque and mark the node as *suspended*; and
- `isEmpty`: return true if there are no ready nodes in the deque (but may contain one suspended future join node).

Just as in ABP, we assume that multiple workers can make calls to a deque concurrently; if more than one worker tries to pop the same element off the deque, one of them succeeds and the other one fails in a constant number of time steps.

Throughout the lifetime of a deque, it can be in one of the following four states:

- ***active:*** it is actively been worked on by a worker;
- ***suspended:*** it is suspended due to a `get` call; every node in the deque is ready, except for the bottommost node, which is the corresponding suspended future join node;

- ***resumable:*** it contains only ready nodes, but it is not actively being worked on by a worker; and

- ***muggable:*** similar to a resumable deque, except that the entire deque can be stolen and resumed.

These states are exhaustive, and a deque can only transition: 1) from active to suspended due to execution of a `get` call, 2) from suspended to resumable due to termination of the future task enabling the join node at the bottom, 3) from resumable to active if the worker who finishes the future task has an empty deque and resumes one of the now-resumable deques suspended with the future handle; 4) from resumable to muggable after a thief steals from it once, and 5) from muggable to active when a thief mugs it and resumes its execution. Since a resumable deque transitions to muggable once it is stolen from, only its top item may be stolen before transitioning. If a thief steals into a muggable deque, it takes the entire deque and resumes its execution from the bottommost node.

The stealable deques belonging to a worker are maintained as a set. Each future handle also maintains a deque set with references to suspended deques, allowing any deques suspended with the handle to be resumed when the future task completes. A deque set supports the following operations:

- `add(`*deq*`):` add deque *deq* into the set;

- `remove(`*deq*`):` remove deque *deq* from the set;

- `removeRandom():` remove and return a deque from the set, chosen uniformly at random; and

- `pickRandom():` return a reference to a deque in the set, chosen uniformly at random (but does not remove it).

We assume that one can make concurrent calls to a given set, and an operation will finish in constant amortized time. When operating on the stealable set of a worker, the worker is always chosen uniformly at random among the P workers. Thus, the contention can be resolved in a constant number of time steps in expectation (e.g., see lemma 6 in [14]). In practice, a set can be implemented as a growable array (performing array doubling when necessary), which maintains a constant amortized insertion cost.

4.2.2 The Algorithm

Algorithm 5 shows the main scheduling loop for ProWS and its helper functions. Ignoring the special handling of future operations in lines 30–37, ProWS behaves the same as ABP. Each worker starts out with one active deque; it operates off the bottom of the deque (line 21 and lines 25–29) and steals when it runs out of work to do (lines 22–23). A worker, when enabling two nodes, pushes the right node (i.e., continuation) first (line 27) and then the left node (i.e., the spawned task) (line 29), which means that the left node gets executed next.

Future operations are handled differently. If the execution of this strand terminates with `get` (lines 30–35) and the corresponding future task f has not terminated, `get` enabled zero nodes. The worker then pushes the corresponding future join node j (the immediate successor of n) onto the bottom of the deque via `pushBottomImplicit` (line 32) and suspends the deque (line 33). The reference to the suspended deque is stored with the future handle of f (line 34) and the worker's active deque is set to *null* (line 35). It will be set to something else after the steal. On the other hand, if the executed strand terminates with `put`, that its corresponding future task f has terminated and all suspended deques stored with f can now be resumed (lines 36–40). At this point, if the worker executing `put` has an empty active deque, it will set its active deque to one of the suspended deques stored with the future handle and resume its execution next (lines 38–40).

Algorithm 5: ProWS: The Main Scheduling Loop

```
   /* w is the executing worker                                          */
1  Function suspend(deq)
2      deq.status = SUSPENDED
3      if dep.IsEmpty() then
4          deq.worker = null
5      else
6          v = ChooseRandomVictim() // can include w itself
7          v.stealable.add(deq)
8          deq.worker = v
   /* w is the executing worker                                          */
9  Function setToActive(deq)
10     if deq.worker then
11         rebalanceStealables(deq.worker)
12         deq.worker.stealable.remove(deq)
13     deq.worker = null // deq is not in any stealable set
14     deq.status = ACTIVE
15     if w.active is not null then
16         freeDeque(w.active)
17     w.active = deq
   /* w is the executing worker                                          */
18 while computation is not done do
19     n = null // n points to next strand to execute
       // w.active points to either null or its active deque
20     if w.active is not null then
21         n = w.active.popBottom()
22     if n is null then
23         steal() // steal returns when work is found
24     else // execute n
25         left, right = execute(n)
26         if right is not null then
27             w.active.pushBottom(right)
28         if left is not null then
29             w.active.pushBottom(left)
           // special case: f is a future handle
30         if n terminated with f.get() then
31             if f is not ready then
                   // j is the future-join node after n
32                 w.active.pushBottomImplicit(j)
33                 suspend(w.active)
34                 f.suspended.add(w.active)
35                 w.active = null
36         else if n terminated with f.put() then
               // Mark every deque in f.suspended RESUMABLE
37             markSuspendedResumable(f.suspended)
38             if w.active is empty then
39                 deq = f.suspended.removeRandom()
40                 setToActive(deq)
```

The implementation of suspend is shown in lines 1–8. Since ProWS may potentially suspend many deques, it takes extra steps to ensure that the number of stealable deques are roughly balanced among workers. Instead of suspending with the current worker w, it chooses a target worker v uniformly at random (which can include w itself) and suspends the deque with v. The reference to v is stored with the suspended deque so that when the deque gets resumed it can be removed from worker v's stealable set.

If the suspended deque contains no ready nodes (line 3) we don't store the deque in any worker's stealable set, as it has nothing to be stolen from. Such a deque, once gets resumed, is inserted into a stealable set of a worker chosen uniformly at random (by `markSuspendedResumable` in line 37).

Finally, a key thing to note in `setToActive` is that it invokes `rebalanceStealables` (line 11), which is invoked whenever w is about to remove a deque from v's stealable set — it randomly chooses another victim v'; if $v = v'$, w is done; otherwise w moves a stealable deque from v' to v if v' has one. Section 4.3 explains why we do such a rebalance.

Algorithm 6 shows the implementation of the steal protocol that a worker w invokes when its deque becomes empty or after it loses its deque due to suspension. The steal function performs steal attempts until w finds work successfully.

When stealing, w chooses a victim v uniformly at random (line 43, which again includes w) and chooses a deque uniformly at random among v's deques (line 44). If the chosen deque is muggable, w takes the whole deque and set it to be its active deque. Otherwise, w steals from the top (line 49). After `popTop`, if the deque runs out of ready nodes, it is removed from v's stealable set and possibly destroyed if there isn't even a suspended future join node at the bottom, such as in the case of resumable deque (line 51). Moreover, `rebalanceStealables` is invoked again. If the deque is resumable and not empty, it is

Algorithm 6: ProWS: The Steal Protocol

```
   /* w is the executing worker                                          */
41 Function steal()
42 |  while true do // steal returns only when work is found.
43 |  |  v = ChooseRandomVictim() ; // can include w itself
44 |  |  deq = pickRandom(|v.active| ∪ |v.stealable|);
45 |  |  if deq is null then continue;// Nothing to steal from v
46 |  |  if deq.status is MUGGABLE then
47 |  |  |  setToActive(deq);
48 |  |  |  break;
49 |  |  n = deq.popTop() ; // deq is suspended or resumable
50 |  |  if deq.isEmpty() then
51 |  |  |  handleEmptyDeque();
52 |  |  |  rebalanceStealables(v);
53 |  |  else if deq.status is RESUMABLE then
54 |  |  |  deq.status = MUGGABLE
55 |  |  if n is not null then
56 |  |  |  if w.active is null then
57 |  |  |  |  w.active = newDeque()
58 |  |  |  w.active.pushBottom(n);
59 |  |  |  break;
```

marked as muggable (line 54). After a successful steal, w may need to allocate a new deque (lines 57 and 58).

4.3 Performance Bounds for Proactive Work-Stealing

This section analyzes ProWS to show that, 1) for a computation with T_1 work and T_∞ span, it executes the computation in expected time $O(T_1/P + T_\infty \lg P)$, and 2) the number of deviations is bounded by $O(PT_\infty^2)$ for a program that uses structured futures and $O(m_k T_\infty + PT_\infty \lg P)$ for a program that uses general futures.

Before we analyze the bounds, we first show that, at any point during the execution, the set of stealable deques are roughly evenly distributed across workers, which we utilize when we discuss the bounds. We use the following lemma on the classic balls-into-bins problem, which is not hard to show (see e.g., [56, Chp. 5]):

Lemma 20. *When m balls are thrown independently and uniformly at random into n bins, the probability that the maximum load is more than $\frac{m}{n} + O(\lg n)$ is at most $1/n$. Similarly, the probability that the minimum load is less than $\frac{m}{n} - O(\lg n)$ is at most $1/n$.*

Lemma 21. *Given P workers and S number of stealable deques in the system, with probability $1 - o(1)$ each worker has at most $S/P + O(\lg P)$ deques.*

Proof Sketch. One can model the number of stealable deques per worker as the classic balls-into-bins problem, where the workers are modeled as bins and the stealable deques are modeled as ball tosses. Our process also includes muggings, however, which changes the size of the stealable sets, and thus the analysis requires additional care.

We model the entire process as two separate ball-toss processes: a ***deque-suspension*** process, where a suspended deque is modeled as a ball toss into a randomly-chosen bin (worker to leave the deque with), and the ***deque-removal*** process, where removing a deque is also modeled as a ball toss into a randomly-chosen bin (worker to remove the deque from). Then the size of a given stealable set is the number of balls resulted from the deque-suspension process minus the number of balls resulted from the deque-removal process. The upper and lower bounds on the maximum and minimum loads in Lemma 20 thus give us the desired bound.

It is not hard to see that the workers from the deque-suspension process is chosen uniformly at random. What remains to be shown is that the same holds true for the deque-removal process. There are a couple ways a deque can disappear from a stealable set: 1) a worker takes the whole deque to resume it (lines 40 and 47); and 2) a deque becomes empty after it is stolen from (lines 50–52). In both cases, we always invoke `rebalanceStealables`: if we are removing a deque from v, we randomly choose a victim v' to move a stealable deque to v. If v' has a stealable deque to move to v, it's as if we removed the deque from v'. If v' does not have a stealable deque, it's as if we first moved the deque to v' and then removed it. Pretending to move a deque from v to v' is ok, since v has a larger stealable set at the

moment, and doing so simply balances the load from a more-loaded worker to a less-loaded one. Even though such load balancing is conditioned on v' not having any deque, doing so does not hurt the bound. □

4.3.1 Bound on Execution Time

Our time bound analysis follows a similar structure to the analysis done in [4] and [99]. We separately bound the number of time steps devoted to various activities: work, steal attempts, and muggings. By bounding how many time steps each activity takes, the final bound arises by summing all the time steps divided by P, the number of workers used. Obviously, the total work is bounded by T_1 time steps.

It remains to bound the number of steal attempt and mugging operations, each taking a constant number of time steps. In the original work-stealing analysis by Arora et al. [4, 5], henceforth referred to as ABP, steal attempts are bounded by a potential function argument that states the following. Assuming there are P deques in the system, after $O(P)$ steal attempts, the overall potential decreases by a constant fraction. This is because, the topmost node in a deque contributes to a constant fraction of the overall potential among nodes within the deque.

More formally, the following lemma is a straightforward generalization of lemma 7 and 8 in ABP [4] which we utilize:

Lemma 22. *Let Φ_i denote the potential at time t and say that the probability of each deque being a victim of a steal attempt is* at least $1/X$. *Then after X steal attempts, the potential is at most $\Phi(t)/4$ with probability at least* $1/4$. □

Effectively, this lemma says that the number of steal attempts is at most $O(XT_\infty)$, since the potential function is a function of T_∞. For ABP, it is always the case that $X = P$, leading to a steal attempts bound of $O(PT_\infty)$.

The ABP analysis cannot be applied to ProWS directly, since 1) ProWS can have more than P deques in the system, and 2) a thief stealing into a muggable deque will resume the bottommost node in the deque instead of the topmost one, which may not contain sufficient amount of potential.

To resolve issue 1), we apply a similar technique to Utterback et al. [99] and divide the computation into two types of phases: a ***steal-bounded phase*** when there are at most $2P$ stealable deques, and a ***work-bounded phase*** when there are more than $2P$ stealable deques. During a steal-bounded phase, by Lemma 21, we know each worker has at most $O(\lg P)$ deques, leading to a steal attempts bound of $O(T_\infty P \lg P)$ by Lemma 22. During a work-bounded phase, the total number of deques in the system is more than $3P$. However, since there are many deques in the system distributed roughly equally among workers, steal attempts are likely to succeed, each followed by a unit of work. Thus, we can bound the steal attempts by $O(T_1)$ during a work-bounded phase. Overall, this leads to an execution time bound of $O(T_1/P + T_\infty \lg P)$.

We still need to resolve issue 2) and in addition bound the time spent on muggings. Recall that in ProWS, we enforce that every resumable deque has to be stolen from once before it becomes muggable. This may seem counter-intuitive — why not simply resume the deque from the bottom if it is already resumable? This steal-before-mug ensures that for each mugging there is a corresponding successful steal on the same deque to amortize against. Doing so prevents the worst case scenario where a deque with a high-potential node on top repeatedly becomes resumable and mugged but never stolen from. This scenario would prevent us from bounding steal attempts that lead to a successful mugging.

Thus, we can also bound the time steps spent on mugging against steals, resulting the following time bound:

Theorem 23. *Consider a computation with T_1 work and T_∞ span. The expected execution time is $O(T_1/P + T_\infty \lg P)$.*

4.3.2 Bounds on Deviations

We first define some notations. Given a computation dag G, we say that u is a ***predecessor*** of v and v is a ***successor*** of u iff there is a directed path from u to v.

We make the following assumption. Let u be a node with two outgoing edges, meaning that u can be a spawn node, a future spawn node, or a future put node. The only way for a future put node to have an out-degree of two is if the corresponding future is multi-touch, which creates a chain of put nodes, each with an out-degree of two.

Given a computation dag G, the ***sequential order*** is a total ordering of nodes in G that arises from the sequential (one-worker) execution. A ***processor order*** of a worker w is the sequence of nodes processed by w in a parallel execution of ProWS. We say $u <_1 v$ if u is before v and $u \prec_1 v$ if u is immediately before v in the sequential order. Similarly, we say $u <_w v$ if u is before v and $u \prec_w v$ if u is immediately before v in the processor order of w. Given this notation, we now formally define ***deviation***:

Definition 24. *Let u and v be two nodes in a dag and $u \prec_1 v$. We say that v is a deviation in the parallel execution if for some worker w that executed v, we have $u \not\prec_w v$.*

Given the definition of SP-dags (Section 2.2), it's not hard to see that for every sync node v in an SP-dag G, there is a corresponding spawn node u. Let $u.lchild$ denote the left child of u and $u.rchild$ denote the right child of u. Similarly, let $v.lparent$ denote the left parent of v and $v.rparent$ denote the right parent of v. Then, let G_{left} be the SP-subdag that consists of the set of nodes x in G such that there is a path from $u.lchild$ to x and from x to $v.lparent$. We say G_{left} is the SP-dag ***enclosed*** by $u.lchild$ and $v.lparent$.[9] We define G_{right} symmetrically. We first show properties of the sequential and parallel executions when scheduled with ProWS.

[9]Recall that each future task is treated as its own SP-dag and thus if a node x in G spawns a future task via `create` none of the nodes belonging to the future task is in G.

Lemma 25. *Given an SP-dag G enclosed by a spawn node u and a sync v, let x be a node in G_{left} and let y be a node in G_{right}. Then, $x <_1 u.rchild <_1 y <_1 v.rparent$. Moreover, $v.rparent \prec_1 v$.*

Proof. Recall from Section 4.1 that, for a program that uses futures, the computation can be modeled as multiple SP-dags connected via create and join edges with each future task modeled as its own SP-dag. We can show that this lemma holds by inducting on the number of independent SP-dags.

For the base case, where the program does not use any futures, the program itself is a single SP-dag, and the statement is true, based on the recursive structural properties of an SP-dag. Now assume we have two SP-dags, one for the main program F and one for a future task G created via invoking `create` in F. When ProWS executes the `create`, it pushes its corresponding continuation strand z onto the bottom of its deque. It then goes on to execute the source node of the SP-dag for G. Since inductively the statement also holds true for the execution for G, z should remain on the deque and can only be popped off to execute after ProWS executes the future put node corresponding to the last strand of G. One can generalize the statement to a program with multiple futures similarly. □

Effectively, this lemma says that the sequential order of ProWS follows the depth-first left-to-right traversal of the dag. Moreover, since for either structured or general use of futures the `create` of a future task f must appear before the corresponding `get` in sequential order, the sequential execution of ProWS can never suspend due to a `get`.

Now we prove lemmas about parallel executions.

Lemma 26. *Let v be a sync node and u its corresponding spawn node. If v is a deviation, then $u.rchild$ must be stolen.*

Proof. Let $u.lchild$ and $u.rchild$ be the left and right child of u. Similarly, let $v.lparent$ and $v.rparent$ be the left and right parent of v. Following Lemma 25, we know that $v.rparent \prec_1$

v. Then we must have $v.lparent \prec_w v$ for some worker w since v is a deviation. Therefore, $u.rchild$ must have already been executed when $v.lparent$ enabled v because $u.rchild$ is a predecessor of v.

For the purpose of contradiction, suppose that $u.rchild$ is never stolen. Consider a worker w_1 that executes u. It pushes $u.rchild$ on the bottom of the deque and continues executing $u.lchild$. Since if $u.rchild$ is never stolen, the only way for it to execute is by popping it off the bottom of the deque. By Lemma 25, we know that sequentially a worker will not pop $u.rchild$ off the deque before executing $v.lparent$.

In a parallel execution, by Algorithm 5, the only thing that can cause w_1 to deviate from the sequential order is if it encounters a `get` that causes it to suspend. In this case, w_1 pushes the join node onto the bottom of its deque and suspends the entire deque. Later, whenever the deque becomes resumable, either $u.rchild$ eventually gets stolen from the top, which contradicts our assumption, or some worker w_2 mugs and resumes the execution starting from the suspended join node. However, that means, w_2 will resume execution and go back to following the sequential order starting from the join node, and thus will not pop $u.rchild$ off the deque before executing $v.lparent$. □

Lemma 27. *If a worker w enables no children after executing the right parent of a sync node, then $w's$ deque is empty.*

Proof. Let v be the sync node and u the corresponding spawn node. By 26, if v is a deviation, $u.rchild$ is stolen. Consider the SP-subdag G enclosed by $u.rchild$ (G's source node) and $v.rparent$ (G's sink node). Then any node in G is executed before $v.rparent$ in any execution. We know w must steal $u.rchild$ or any node in G, otherwise there is no possibility for w to process $v.rparent$. Suppose the deque is not empty after executing $v.rparent$. Let z be bottommost node on the deque. We must have z outside G since any node in G has been executed. Furthermore, $w's$ deque is empty when w performs the steal. Then everything in

the deque afterwards is a descendent of $u.rchild$. So z can only be a node in a future dag spawned by G.

Worker w will turn to the future subdag immediately after executing the corresponding future spawn node. There are two ways that w can resume the execution of G: (1) the future completes, or (2) $w's$ deque is empty again and it performs a steal targeting a node in G. In both cases, z cannot be on $w's$ deque, which contradicts our supposition. □

At a high-level, we bound the number of deviations as follows. We define the notion of "traces" that divide the sequence of nodes executed by a worker based on the types of nodes. We then show that only the first node in a trace can incur a deviation. Lastly, we show that, such a node is either the direct result of a successful steal or can be amortized against a successful steal.

Definition 28. *Consider a sequence of nodes processed by w, which we then separate into a set of **traces**, where each trace begins with one of the following nodes: (1) a sync node, (2) a node that gets executed immediately after w performs a successful steal, and (3) a node that gets executed immediately after w performs a successful mugging.*

Observation 1. *Given a node n in the dag, n can be one of the following:*

1. *n is a **regular node**: n has one child in the dag, and the child has only n as a parent;*
2. *n is a **spawn node**: n has two children in the dag, where each child has only one parent;*
3. *n is a **future put node**: n can have either one child (single touch future) or two children (chain of put nodes for multi-touch futures);*
4. *n is a **parent of a sync node**: n has one child, where the child has two parents;*
5. *n is the **local parent of a future join node**: n has one child, where the child has two parents.*

It can be seen that these types are exhaustive by enumerating all the possible combinations of the out-degree of n and in-degree of n's children. Note that we can never have a spawn node n leading to a child who is a sync node or join node — the act of invoking `get` or `sync` terminates the current strand and creates a new node with an in-degree of two. Thus, a `get`/ `sync` that immediately follows a `spawn`/ `create` will have a node inserted between them.

Lemma 29. *Consider a sequence of nodes executed by a worker w during parallel execution scheduled using ProWS, which we separate into traces according to Definition 28. For a given trace $t = (n_1, n_2, \ldots, n_l)$, only n_1 can be a deviation.*

Proof. Let $s = (n_1, \hat{n_2}, \ldots, \hat{n_l})$ be the sequence of l nodes that starts with n_1 in sequential execution. We show that $n_i = \hat{n_i}$ for $i = 2 \ldots l$ by inducting on the length of the trace and argue that either processor w behaves exactly as sequential execution, or the trace ends.

Inductively, assume $n_i = \hat{n_i}$ for $i = 2 \ldots j-1$ and w behaves exactly the same as the sequential execution up to that point (i.e., each n_i enabled exactly the same nodes as $\hat{n_i}$). Now we consider n_j. Based on Observation 1, n_j can be one of the following. **regular node:** n_j must enable its only child and execute it next, just as in sequential execution.

spawn node: n_j must enable both children, executing the left one and pushing the right one onto the deque, just as in sequential execution.

future put node: if n_j is a put node for a single-touch future or one at the end of a put chain, then n_j can either enable nothing or the corresponding future join. If n_j enables nothing, this is the same as sequential execution, and w either pops its bottom deque (which leads to the same n_{j+1} by inductive hypothesis), or trace t ends at n_j if w's deque is empty, since the next node has to follow from either a successful steal or mugging. If n_j enables the corresponding future join node j, that means the local parent of j executed and couldn't enable j and thus pushed j onto the bottom of some (now suspended) deque. Even though

n_j enabled j, note that in ProWS, it does not push j onto w's deque. Instead, it simply marks the deque as resumable.

On the other hand, n_j can be a put node for a multi-touch future that enables the next put node and may or may not enable the corresponding future join node. Enabling the next put node is exactly the same as sequential execution, and whether the corresponding future join node is enabled or not does not matter, following similar argument as above.

parent of a sync node: If executing n_j enables this sync node, then trace t ends at n_j (by the definition of the trace). If in both sequential and parallel executions, n_j enables no child, then w tries to pop a node off the bottom of its deque, which leads to the same n_{j+1} by the inductive hypothesis. On the other hand, if n_j enables no child but in sequential execution, $\hat{n_j}$ enables the sync node, then n_j must be the right parent of the sync node. Then by Lemma 27 w's deque must be empty and thus trace t ends at n_j.

local parent of a join node: In the sequential execution, since the local parent always enables the join node, $\hat{n}_{j+1}$ will be the join node. In the parallel execution, either n_j also enables the join node, which means $n_{j+1} = \hat{n}_{j+1}$, or it enables no child. In the latter case, w will push the join node onto the bottom of the deque and suspend the deque, which mean trace t ends at n_j because the next node has to follow from either a successful steal or mugging. □

Finally, a key theorem to bound the number of deviations:

Theorem 30. *Given an execution of ProWS. Let n be the number of successful steals in the execution. Then, the number of deviations is $O(n)$.*

Proof. From the definition of traces and Lemma 29, we know a deviation may only occur at the beginning of a trace. Each trace begins with a sync node, a stolen node, or a node processed after a successful mugging. Thus, the number of deviations is bounded by the sum of the numbers for deviated sync nodes, successful steals, and muggings.

From Lemma 26, we know each deviation at a sync nodes has a unique corresponding stolen node, thus we can bound the number of deviated sync nodes by the number of successful steals. Also recall that in ProWS, a resumable deque has to be stolen from successfully before it becomes muggable. Thus, the number of successful muggings is also bounded by the number of successful steals. Thus, the total number of deviations is bounded by $O(n)$, where n is the number of successful steals. □

Given Theorem 30, we can now bound the deviations by bounding the number of successful steals, which is less than the number of steal attempts during the computation. Recall Lemma 22, which effectively states that the number of steal attempts can be bounded by $O(XT_\infty)$, when a deque can be stolen into with probability at least $1/X$. We provide a bound on $1/X$ by bounding the the maximum number of stealable deques possible during the execution.

Lemma 31. *Given a computation that uses structured futures scheduled using ProWS, there can be at most $O(PT_\infty)$ stealable deques during execution.*

Proof. In the case of the structured use of futures, the stealable deques can only include suspended deques. Recall that a structured use of futures is restricted to single touch and no race on the handle (i.e., the future spawn node has a directed path to the local parent of the join node). Due to the former, there can exist only one suspended deque per future handle f. Moreover, due to the directed path, the continuation of the future spawn node that spawned f must be stolen in order for a corresponding `get` on f to suspend. Thus, whenever a worker w eventually executes the `put` that completes f's corresponding future task, w's deque must be empty. By Algorithm 5, w will then resume the single deque suspended with f, making it active, and thus there cannot be resumable or muggable deques in the stealable set.

Whenever a worker has to suspend a deque due to `get`, the corresponding future task f is either being actively worked on by another worker (due to eager evaluation), or f is also

suspended because f itself invoked a `get` on a different future, creating a chain of suspended future tasks. Such a chain has length at most T_∞ (as any chain in the dag). Moreover, at least one worker is working on the future task at the beginning of the chain. Therefore, there can be at most $O(PT_\infty)$ suspended (stealable) deques. □

Lemma 32. *Given a computation that uses general futures scheduled using ProWS, there can be at most m_k stealable deques during execution.*

Proof. By Algorithm 5 lines 30–35, a deque can only suspend when encountering a `get` (future touch). When a future touch node n causes a deque to suspend, no descendant of n can execute until the deque becomes active again. By definition, since there can be at most m_k number of future touches executing in parallel, this leads to a maximum number of m_k stealable deques at any given time. m_k. □

Finally, we can prove the following deviation bounds:

Theorem 33. *Given a computation that uses structured futures with span T_∞ and scheduled using ProWS on P workers, the number of deviations is $O(PT_\infty^2)$ in expectation.*

Proof. By Lemma 31, we know the maximum number of deques possible during execution is $O(PT_\infty)$. By Lemma 21, each worker can have up to $O(T_\infty + \lg P)$ deques. Thus, a deque is stolen into with probability of at least $O(\frac{1}{PT_\infty + P \lg P})$. Then by Lemma 22, the steal attempts across the computation is at most $O(PT_\infty^2 + PT_\infty \lg P)$, or $O(PT_\infty^2)$ assuming $T_\infty = \Omega(\lg P)$, which is likely the case. □

Theorem 34. *Given a computation that uses general futures with span T_∞ and scheduled using ProWS on P workers, the number of deviations is $O(m_k T_\infty + PT_\infty \lg P)$ in expectation, where m_k is the maximum number of future touches that are logically parallel.*

Proof. By Lemmas 32 and 21, we similarly derive the probability that a deque is stolen into to be at least $O(\frac{1}{m_k + P \lg P})$. Then by applying Lemma 22 we obtain the bound. □

Chapter 5

Race Detection for General Futures

Futures [36] provide an elegant means to express parallelism in task parallelism. However, only a few prior works [2, 90, 98] exist that study the problem of race detecting programs that use futures. Unlike programs that utilize only fork-join or pipeline parallelism, the use of futures can generate arbitrary dependences due to the fact that the joining of future tasks can occur at arbitrary program points. Consequently, a program that uses futures no longer has the same structural properties that enable efficient race detection algorithms. The lack of structural properties has a few implications. First, it no longer suffices to store only a constant number of accessors per memory location. Instead, the number of readers per memory location can be large — all the readers must be recorded until a sequential writer comes along. Second, the reachability can no longer be maintained and queried as efficiently using two total orders.

In this chapter, we examine the race detection problem with ***general futures***.[10] The execution of a parallel program using general futures can be modeled as a ***nearly series-parallel dag (NSP-dag)***, that is a dag formed by a set of SP-dags connected by a set of

[10]The general use of futures was previously defined in Section 4.1.2 with two restrictions: constant number of touches per future and forward pointing. However, the algorithm (F-Order) proposed in this chapter does not rely upon these restrictions.

arbitrary additional edges, representing the arbitrary dependencies that arise from the use of futures.

The state-of-the-art algorithm [2] that targets NSP-dags provides an execution time bound of $O(T_1 + k^2)$, where k is the number of future operations. However, this algorithm must execute the program sequentially during race detection. The requirement of sequential execution is not just an implementation artifact but fundamental to how the algorithms maintain reachability. As the computation dag unfolds dynamically during program execution, the reachability maintenance data structure must maintain reachability of all accesses occurred thus far. In the state-of-the-art sequential algorithm, the reachability data structure can only be maintained correctly assuming a particular traversal order of the computation dag, which only the sequential execution guarantees. If the dag unfolds in any other topological traversal order (which occurs during parallel executions), the reachability data structure proposed by these prior algorithms no longer work correctly.

In this work, we propose F-Order, the first known *parallel* race detection algorithm that race detects a program with general futures while executing the program in parallel (Sections 5.2 and 5.3). We show that, given a computation with T_1 work and T_∞ span, our race detection algorithm runs in time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(\lg k + \lg r \lg \hat{k}))$ on P processors, where k is the number of future operations, r is the maximum number of readers per memory location, and $\hat{k}$ is the maximum number of future operations done by a single future task, which is usually small (Section 5.4).

To put this bound into perspective, a provably efficient parallel scheduler can execute a baseline program (i.e., no race detection) in time $O(T_1/P + T_\infty)$ [5]. Our race detection algorithm incurs additional $O(k^2)$ work, like the state-of-the-art sequential algorithm [2], but this additional work can be parallelized. It also incurs a multiplicative overhead of $\lg \hat{k}$ on the work term (which is small) and a multiplicative overhead of $(\lg k + \lg r \lg \hat{k})$ on the span term.

We have implemented and empirically evaluated a prototype system based on F-Order (Section 5.5). The empirical results indicate the reachability component incurs little overhead and that the race detection obtains similar scalability as the baseline. Moreover, we have compared our parallel race detector against FutureRD, the state-of-the-art sequential race detector for futures [96, 98]. Empirical results indicate that, even though our parallel race detector incurs higher overhead on one-core execution, the fact that we can race detect while executing the program in parallel quickly pays off in absolute execution times.

5.1 Nearly Series-Parallel Dag

When a program uses both fork-join and general futures, the execution generates a ***nearly series-parallel dag (NSP-dag)*** $G_N = (D_{sp}, E_{non})$, a dag formed by a set D_{sp} of SP-dags connected by a set E_{non} of ***non-SP*** edges (create and get edges). Since each future task may contain fork-join parallelism, the execution of a future task can be modeled as its own SP-dag. If the future task executes a `create`, the spawned-off future task is a separate SP-dag, connected via non-SP edges.

For ease of description, we say the following types of nodes in the dag are special: spawn, sync, create, join, and put nodes.[11] All other nodes are ***common nodes***. Furthermore, we say create and put nodes are ***non-SP nodes***, which are special in that they have an outgoing non-SP edge. If there is a path from u to v, and u is either a create or put node, we say u is a ***non-SP ancestor*** of v.

5.2 Overview of F-Order

This section provides an overview of F-Order. As mentioned at the beginning of this chapter, the use of futures generates non-SP edges that form arbitrary dependences and thus lack the structural properties that fork-join parallelism enjoys, which brings challenges to the race

[11] These types of nodes are previously defined in Sections 2.2 and 4.1

detection problem for futures. We now discuss each of the challenges, the intuitions behind F-Order, and how F-Order addresses the challenges.

5.2.1 Access History in F-Order

For fork-join (and pipeline) parallelism, due to the nice structural properties of SP-dags, it is sufficient to store only the "left-most" ("down-most") and "right-most" readers per memory location during parallel execution [55]. One can prove that a reader omitted by the access history can race with a writer if and only if the writer also races with either the left-most (down-most) or the right-most reader. Thus, we do not miss a race by omitting such a reader in the access history. For a program that uses futures, however, we no longer have the same structural properties — there are no clear "left-most" and "right-most" readers that one can store to subsume potential races with other readers. Thus we must store all readers encountered until a sequential writer comes along.

How F-Order maintains access history follows the same strategy as the prior state-of-the-art sequential algorithm [2]. For each memory location l, F-Order stores a ***last writer***, `writer`(l), which is the last node that wrote to l, and a list of readers `reader-list`(l) that read l since `writer`(l). Whenever a node r tries to read a memory location l, F-Order performs a reachability query between r and `writer`(l) to see if r races with the last writer. If so, a race is reported. If not, r is added to `reader-list`(l). Whenever a node w writes to a memory location l, F-Order checks w against all the readers in `reader-list`(l) and the last writer `writer`(l). If any of the `reader-list`(l) or `writer`(l) is logically in parallel with w, a race is reported. Otherwise, F-Order empties the `reader-list`(l) and sets `writer`(l) to be w. As argued by Agrawal et al. [2], we won't miss any races by emptying `reader-list`(l) because any future access that races with a node in `reader-list`(l) must also race with w. The fact that the prior algorithm executes the program sequentially

and F-Order executes it in parallel does not change the correctness argument, so long as F-Order synchronizes the access history data structure correctly.

Agrawal et al. [2] also show that that the total number of reachability queries per reader is bounded by two. F-Order provides the same bound on the number of reachability queries per reader. Since F-Order executes in parallel, however, we must also consider how the reachability queries impact the span, which we discuss in Section 5.4.

5.2.2 Reachability Maintenance in F-Order

The Challenges. A computation that uses futures can be modeled as a NSP-dag, consisting of a set of SP-dags connected via non-SP edges. Given two nodes, if they are connected by only SP edges, one can perform a reachability query on them efficiently by applying the reachability maintenance algorithm used for fork-join parallelism from prior work [9, 29, 97]. The challenge is to handle the reachability queries efficiently when the two nodes are possibly connected in part by non-SP edges.

The state-of-the-art sequential algorithm [2] encodes the reachability that arises due to non-SP edges explicitly using an auxiliary graph $\mathcal{R}$. Unfortunately, the maintenance of $\mathcal{R}$ heavily depends on traversing the computation dag in a left-to-right depth-first fashion (i.e., executing the dag sequentially). Thus, this prior algorithm simply cannot be parallelized, since a parallel execution can traverse the dag in any order (as long as it is a topological sort of the dag), which breaks the invariants required by $\mathcal{R}$ to keep track of reachability correctly. Thus, the reachability maintenance in F-Order has to use an entirely different strategy.

The Intuitions. Based on how we model the computation, each future task forms its own SP-dag, and different SP-dags can only be connected via non-SP edges. That means, if two nodes are in series and connected by only SP edges then they must belong to the same SP-dag, and one can utilize a prior parallel algorithm such as WSP-Order [97] to correctly answer reachability queries between them. WSP-Order cannot encode the reachability of two

nodes correctly if they are connected via non-SP edges, regardless of whether they belong to the same SP-dags or not. Thus, we need some other means to encode reachability between two nodes if they are connected via non-SP edges.

The key observation is as follows. Given two nodes u and v connected via non-SP edges, some node w must exist in the path between u and v, where w is a non-SP ancestor of v that is in the same SP-dag as u. That is, the prefix of the path (from u to w) contains only SP edges, and the suffix (from w to v) containing at least one non-SP edge (outgoing from w). Thus, given two nodes u and v possibly connected in part by non-SP edges, we can query their reachability efficiently if we can quickly determine if such an ancestor w exists, who is 1) an non-SP ancestor of v in the same SP-dag as u and 2) in series with u via only SP edges.

The FOM Data Structure. To quickly determine whether such an non-SP ancestor w of v exists, F-Order employs an enabling data structure called the ***Future Order-Maintenance*** (or ***FOM*** for short) data structure per node in the NSP-dag. An FOM data structure for v stores all v's non-SP ancestors, organized into groups, where each ***group*** contains non-SP ancestors from the same SP-dag.

Upon execution of a node v, its FOM data structure will be complete and its content fixed because v's FOM data structure contains only v's non-SP ancestors, and these must have already been discovered by the time v executes as the scheduler guarantees that a node cannot execute until all its ancestors have executed. If v accesses some memory location that some node u accessed previously, F-Order queries v's FOM data structure to find the group g that holds the non-SP ancestors from the same SP-dag as u. F-Order then checks with g to determine whether some w exists that is reachable from u. If a node is in the group g, by definition it is in the same SP-dag as u. Then, we simply need to check if w is in series with u.

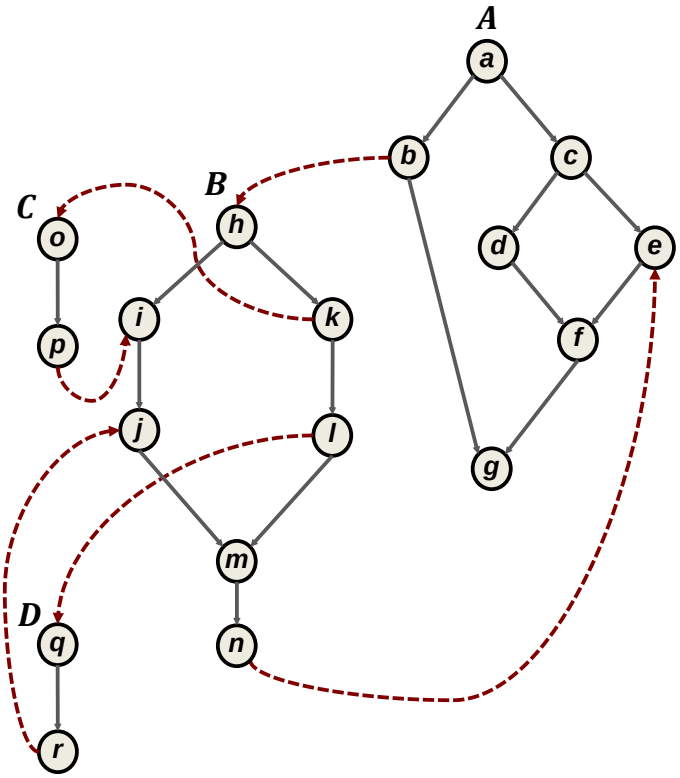
A naive implementation would be to check reachability against every node in g, taking time linear in the size of g. Ideally, we would like to quickly eliminate non-viable candidates in the group and avoid querying every single node in the group. The key insight of F-Order involves identifying the correct auxiliary data to store with each non-SP ancestor in a group so that the process of elimination can be done quickly.

It turns out that, to perform the process of elimination within a group, we simply need to organize nodes in a given group as follows. First, store the nodes in the ***English*** order [62], that corresponds to the depth-first-left-to-right traversal of nodes in the corresponding SP-dag. The English order is well defined among nodes in the same group, because a group contains only nodes from the same SP-dag with no non-SP edges. Second, with each node w in the group, additionally store w's ***furthest descendent*** that is also within the same group; that is, some node z that is reachable from w that is also in the group such that no other node y in the group is reachable from z (formally defined in Section 5.3). We will elaborate on the detailed construction of FOM data structures and discuss how F-Order uses them to perform reachability queries in Section 5.3.

5.2.3 An Illustrating Example

Figure 5.1 shows the static snapshot of an NSP-dag with all its FOM data structures shown. Note that the parallel execution unfolds the NSP-dag dynamically, revealing each node as it becomes ready (i.e., all its ancestors have executed). Nevertheless, as discussed earlier, the content of an FOM data structure for node v is fixed by the time v executes, so this dynamic unfolding of the dag does not change the content of the FOM data structures shown.

For now, we shall focus solely on the organization of an FOM data structure and how we use it to perform reachability queries. Take node f: it contains multiple non-SP ancestors. They are grouped into four entries in f's FOM data structure, as they respectively belong to four different SP-dags. In particular, f has nodes k, l, and n as its non-SP ancestors,



node	FOM
a	{}
b	{A: b(b)}
c	{}
d	{}
e	{A: b(b)}, {B: k(n), l(n), n(n)}, {C: p(p)}, {D: r(r)}
f	{A: b(b)}, {B: k(n), l(n), n(n)}, {C: p(p)}, {D: r(r)}
g	{A: b(b)}, {B: k(n), l(n), n(n)}, {C: p(p)}, {D: r(r)}
h	{A: b(b)}
i	{A: b(b)}, {B: k(k)}, {C: p(p)}
j	{A: b(b)}, {B: k(l), l(l)}, {C: p(p)}, {D: r(r)}
k	{A: b(b)}, {B: k(k)}
l	{A: b(b)}, {B: k(l), l(l)}
m	{A: b(b)}, {B: k(l), l(l)}, {C: p(p)}, {D: r(r)}
n	{A: b(b)}, {B: k(n), l(n), n(n)}, {C: p(p)}, {D: r(r)}
o	{A: b(b)}, {B: k(k)}
p	{A: b(b)}, {B: k(k)}, {C: p(p)}
q	{A: b(b)}, {B: k(l), l(l)}
r	{A: b(b)}, {B: k(l), l(l)}, {D: r(r)}

Figure 5.1: An example of a NSP-dag with every node's FOM data structure shown. In this NSP-dag, four SP-dags exist, ID'ed as A, B, C, and D, with A being the main SP-dag and the others being the spawned future tasks. The non-SP edges are shown as thick dashed edges. Each node has its own instance of FOM data structure, containing entries of $\{key : value\}$ pairs, where the key is the ID of an SP-dag and the corresponding $value$ is a set of non-SP ancestors from the SP-dag. The parentheses next to each non-SP ancestor shows its furthest descendant in the group.

all from SP-dag B. Thus, its FOM data structure contains an entry with group keyed by B, and the corresponding value is a list of non-SP ancestors ordered in their English order. (Note that the nodes in a given SP-dag are labeled alphabetically according to their ordering in the English order.)

Say f is being executed, and F-Order wants to check if f is reachable from node i. Since i belongs to SP-dag B, F-Order checks f's FOM data structure for the group indexed with B, which returns the list k, l, and n (with every node having n as its furthest descendant). Since n is reachable from i, F-Order concludes that f is reachable from i. In this case, the

group for B is small, containing only three nodes. However, the size of a group can be larger, and ideally we want F-Order to quickly home in on n and not check i against every node in the group. This is where the English ordering and the auxiliary data of furthest descendants become useful, which we discuss in Section 5.3.

5.3 Details of F-Order and Its Correctness

This section presents the full detail of F-Order and its correctness proof. F-Order consists of two parts: a ***construction*** algorithm that builds and maintains the FOM data structure for each node and a ***reachability-query*** algorithm that checks whether a given pair of nodes are reachable from one another. We discuss each in turn. Throughout the section, we shall refer back to Figure 5.1 as an illustrating example.

Notations. Given an NSP-dag $G_N = (D_{sp}, E_{non})$, which consists a set of SP-dags connected via non-SP edges, we assume each SP-dag $d \in G_N$ is assigned with a unique integer identifier, denoted as $SP(d)$. Given a node u, we overload the notation and use $SP(u)$ to denote the ID of the SP-dag containing u. Given two nodes u and v such that $u \rightsquigarrow v$, We use $u \rightsquigarrow_{sp} v$ if the path comprises only SP edges and $u \rightsquigarrow_{nsp} v$ if the path comprises any non-SP edge. If u and v are in the same SP-dag $d \in D_{sp}$, we say $u \prec_d v$ iff $u \rightsquigarrow_{sp} v$; we say $u \parallel^d_{left} v$ iff node u is ***left of*** v in SP-dag d, where u is in the left subdag of d and v is in the right subdag of d. Both notations $\prec_d$ and $\parallel^d_{left}$ are only applicable to nodes in the same SP-dag d. They specify the English order in that, if u is before v in the order, then either $u \prec_d v$ or $u \parallel^d_{left} v$. For instance, in Figure 5.1, $c \prec_d g$ and $b \parallel^d_{left} e$. On the other hand, h and g cannot be related using these operators.

5.3.1 Construction of FOM Data Structures

As the NSP-dag unfolds, nodes become ready and get executed. When a node v executes, F-Order constructs an FOM data structure for v, denoted as $v.fom$, whose content is complete at the beginning of v's execution to allow for reachability queries for memory accesses performed by v. An FOM data structure is organized as a hash table, hashing $SP(d)$ to its corresponding group that stores all of v's non-SP ancestors that belong to the SP-dag d.

For a given group g, an element e in g has two fields: $e.node$ and $e.desc$. Field $e.node$ stores the actual non-SP ancestor. Field $e.desc$ stores the ***furthest descendent*** of $e.node$ in g. We say a node v is the furthest descendent of $e.node$ in g iff (1) $e.node \preceq_d v$, (2) there exists an element x in g such that $x.node = v$, (3) for any other element y in g, we have $v \not\prec_d y.node$. Intuitively, the furthest descendant of $e.node$ is another non-SP ancestor of v stored in the same group g that is a descendant of $e.node$ and $e.node$ has no other descendant in the group that is further out.

Properties of an FOM data structure. An FOM data structure maintains the following properties.

1. For a non-SP node w such that $w \preceq v$, there exists a group containing element x in $v.fom$ such that $x.node = w$.

2. Given two elements x and y in group g, $x.node$ and $y.node$ are in the same SP-dag d. If $x.node \prec_d y.node$ or $x.node \parallel^d_{left} y.node$, then x is before y in g.

3. Given an element x in group g, its furthest descendant field is maintained properly. That is, (1) $x.node \preceq_d x.desc$, (2) there exists an element y in g such that $y.node = x.desc$, (3) for any other element z in g, we have $x.desc \not\prec_d z.node$.

Property 1 states that, given a node v in the NSP-dag, $v.fom$ has all v's non-SP ancestors. Property 2 states that the elements in a group are stored in the English order. Since a

group stores only nodes from the same SP-dag, their relationships ($\prec_d$ or $\|^d_{left}$) can be maintained and queried efficiently using the prior parallel algorithm WSP-Order [97] designed for fork-join parallelism. Property 3 states that the furthest descendents for every element is maintained correctly.

The construction algorithm in F-Order assumes the following helper functions that operate on FOM data structures.

- `FOM-Insert(`fom, v`)`: Given an instance fom and a non-SP node v, `FOM-Insert` returns a new instance of FOM created by copying over the content of fom and inserting v into the appropriate group stored in fom.

- `FOM-Merge(`fom_1, fom_2`)`: Given two instances fom_1 and fom_2 of FOM, `FOM-Merge` returns a new instance of FOM created by merging the contents of fom_1 and fom_2.

Algorithm 7: F-Order: Construction

```
1  Function CommonOrSpawn(v)
2  |  v.fom = v.parent.fom
3  Function Sync(v)
4  |  let u be the corresponding spawn node of v
5  |  if u.fom ≠ v.lparent.fom and u.fom ≠ v.rparent.fom then
6  |  |  v = FOM-Merge(v.lparent.fom, v.rparent.fom)
7  |  else if u.fom ≠ v.lparent.fom then
8  |  |  v.fom = v.lparent.fom
9  |  else
10 |  |  v.fom = v.rparent.fom
11 Function CreateOrPut(v)
12 |  v.fom = FOM-Insert(v.parent.fom, v)
13 Function Join(v)
14 |  v.fom = FOM-Merge(v.local.fom, v.future.fom)
```

Algorithm 7 shows the pseudocode of the construction algorithm. F-Order constructs FOMs for all nodes by propagating the presence of non-SP nodes (i.e., create or put) to all its descendents during parallel execution. The algorithm initializes an empty instance of FOM for the source node of the main dag since it doesn't have any ancestor. Subsequently, it

executes nodes of the dag in any valid order: a node can be executed when all it's ancestors have finished executing. As each node v executes, the algorithm calls different functions based on v's node type.

If v is a common or spawn node, v has only one parent $v.parent$. Any non-SP ancestor of $v.parent$ (possibly including $v.parent$ itself) is also a non-SP ancestor of v. Thus, the algorithm sets $v.fom = v.parent.fom$ (line 2).

If v is a create or put node, besides inheriting all its parent's non-SP ancestors, v also needs to insert itself into its own FOM data structure. Therefore, the algorithm invokes `FOM-Insert` to insert itself into $v.parent.fom$ (line 12), which implicitly creates a new instance of FOM that copies the content from $v.parent.fom$ and inserts itself into the appropriate group.

If v is a join node, then v has two parents: a local parent $v.local$ and a future parent $v.future$. Then the algorithm creates a new instance of FOM by merging $v.local.fom$ and $v.future.fom$ (line 14).

Finally, if v is a sync node, then v has two parents: a left parent $v.lparent$ and a right parent $v.rparent$. However, it is not always necessary to merge $v.lparent.fom$ and $v.rparent.fom$. By the structural properties of an SP-dag, we know that a sync node v has a corresponding spawn node u and two SP-subdags in between: the left subdag G_L and the right subdag G_R. The source node of G_L inherits $u.fom$, which can only change when G_L contains either a create node or a join node (i.e., with an incoming put edge). If G_L does not contain either create or join node, $v.lparent.fom$ remains the same as $u.fom$. Similarly, the same thing holds for G_R and $v.rparent.fom$. The merge is only necessary when both $v.lparent.fom$ and $v.lparent.fom$ have changed compared to $u.fom$. Thus, the algorithm checks for whether both have changed, and if so calls `FOM-Merge` (line 6). Otherwise, if only one changed, $v.fom$ inherits the one that has changed (lines 8 and 10). If neither has changed, it doesn't matter which one $v.fom$ inherits from. Take node f in Figure 5.1 for

instance: its corresponding spawn node is c and only the right subdag (i.e., e) has its FOM data structure changed from c. Thus, f simply inherits its FOM data structure from its right parent e. The node g, on the other hand, constructs its FOM data structure by merging the FOM data structures from both of its parents.

For performance reasons, it is important that F-Order calls `FOM-Merge` only when both subdags execute nodes that cause their respective FOM data structure to change, a fact that we use when proving the performance bound of F-Order in Section 5.4.

Lemma 35 states that Property 1 always holds for an FOM data structure:

Lemma 35. *Given a node v, F-Order constructs a FOM instance $v.fom$ that stores all the non-SP ancestors (i.e., future create and put nodes) of v (**Property 1**).*

Proof Sketch. One can show this inductively by the nodes executed during parallel execution: provided that the FOM instance(s) of v's parent(s) satisfy Property 1, the construction of v's FOM also satisfies Property 1. □

To show that the construction algorithm satisfies the Properties 2 and 3, we need to examine the `FOM-Insert` and `FOM-Merge` in more detail. Due to space constraints, we discuss what these functions do at a high-level and omit the full pseudocode.

Insert Operation. At `FOM-Insert(`fom`, `v`)`, we create a new FOM fom_{new} by copying the content of fom into fom_{new}. Then we check if fom_{new} contains a group g with $SP(v)$. If so, we call `Group-Insert(`g`, `v`)`, which returns a new group g_{new} with a copy of g's content and v added, and we replace g with g_{new} in fom_{new}. If not, we simply create a new empty group g_{new} with v added, and add g_{new} to fom_{new}.

Algorithm 8 shows the helper function `Group-Insert`, which uses linear search to find the correct position of v in g_{new}, which keeps Property 2. Furthermore, for a node u in g_{new} such that $u \prec_d v$, v checks its furthest descendent to see if it should be replaced with v; if so, update it in g_{new} (line 23). For any u' that is positioned after v in g_{new}, v cannot be

Algorithm 8: Helper Function: Group-Insert

```
15 Function Group-Insert(g, v)
16     g_new = new Group() // create a new group
17     i = j = 1
18     while j ≤ g.length do
19         x = g[j++] // jth group element in g
20         if x.node ≺_d v ∨ x.node ||^d_left v then
               // copy constructor copying the content of x into y
21             y = new Group-Element(x)
               // update furthest descendant if necessary
22             if x.desc ≺_d v then
23                 y.desc = v
24             g_new[i++]=y // insert group element y into g_new
25         else break // found the right position for v
26
27     g_new[i].node = g_new[i].desc = v
28     i = i + 1
       // copy the remaining elements of g into g_new
29     while j ≤ g.length do
30         g_new[i++] = new Group-Element(g[j++])
31     return g_new
```

u''s furthest descendant as either $v \prec_d u'$ or $v \parallel^d_{left} u'$ and thus we simply copy over the rest (line 30).

We can conclude the following lemma for `FOM-Insert`:

Lemma 36. *Given a FOM fom that satisfies Properties 2 and 3,* `FOM-Insert(`fom`,`v`)` *returns a new FOM with the content of fom and v inserted that satisfies Properties 2 and 3.*

Merge Operation. `FOM-Merge` is used in the construction algorithm to merge two FOM instances. At `FOM-Merge(`fom_1`, `fom_2`)`, we create a new FOM fom_{new}. We first iterate through groups in fom_1 and insert them into fom_{new}. We then iterate through groups in fom_2. For each group g_2 in fom_2, we check if some group g_1 with $SP(g_2)$ already exists fom_{new}. If so, we call `Group-Merge(`g_1`, `g_2`)`, which returns a new group g with merged content, and we replace g_1 with g in fom_{new}. If not, we simply insert g_2 into fom_{new}.

Algorithm 9: Helper Function: Group-Merge

```
32 Function Group-Merge(g_1, g_2)
33     g_new = new Group() // create a new group
34     i = j = k = 1
35     while i ≤ g_1.length ∧ j ≤ g_2.length do
36         x = g_1[i] // the ith group element in g_1
37         y = g_2[j] // the jth group element in g_2
38         if x.node ≺_d y.node ∨ x.node ∥^d_left y.node then
39             z = new Group-Element(x)
40             i = i + 1
41         else if x.node = y.node then
42             z.node = x.node
43             if x.desc ⪯_d y.desc then
44                 z.desc = y.desc
45             else z.desc = x.desc
46             i = i + 1; j = j + 1
47         else
48             z = new Group-Element(y)
49             j = j + 1
50         g_new[k++] = z // insert group element z into g_new
       // copy the remaining elements of g_1 or g_2 into g_new
51     while i ≤ g_1.length do
52         x = g_1[i++]
53         g_new[k++] = new Group-Element(x)
54     while j ≤ g_2.length do
55         y = g_2[i++]
56         g_new[k++] = new Group-Element(y)
57     return g_new
```

Algorithm 9 shows `Group-Merge`, which merges two groups while maintaining the English order during merge, and its operation is akin to the merge step in merge sort. Using a similar correctness proof as merge sort, we can conclude the following lemma:

Lemma 37. *Given fom_1 and fom_2 that satisfy Property 2,* `FOM-Merge(`fom_1, fom_2`)` *returns a new FOM instance with the merged content of fom_1 and fom_2 that satisfies Property 2.*

What is not obvious is that Property 3 is also maintained by `Group-Merge`. Consider the process of merging two groups g_1 and g_2. Given an element x in g_1, $x.desc$ stores the furthest descendent of $x.node$ in the scope of g_1. However, it is possible that there exists a node u in g_2 such that $x.desc \prec_d u$, which means that u should become the new $x.desc$ after merging g_1 and g_2 into g_{new}. It may seem that `Group-Merge` needs to check $x.desc$ against every single node in g_2. It turns out that it is sufficient to only check $x.desc$ against $y.desc$ of an element y in g_2 such that $x.node = y.node$, and during the merge we are guarantee to compare x against y for $x.node = y.node$ (lines 41–46). Lemma 38 states that doing so is sufficient to maintain Property 3.

Lemma 38. *Given two groups g_1 and g_2 that satisfy Property 3,* `Group-Merge` *merges the content of g_1 and g_2 into g_{new} while maintaining Property 3 for g_{new}.*

Proof. Given an element x in g_1, say there exists a node u stored in g_2 such that u is the new furthest descendent of $x.node$. Then, we have $x.node \prec_d u$. Also, suppose g_2 is maintained by $v.fom$. Then we have $u \preceq v$ since all the nodes stored in $v.fom$ are v's ancestors, which leads to $x.node \prec v$. By Property 1, we know $v.fom$ stores all of v's non-SP ancestors. Thus, $x.node$ must be stored in some group of $v.fom$. Since $x.node$ and u are in the same SP-dag, we are guaranteed that $x.node$ is also in g_2. Thus, there must exist an element y in g_2 such that $y.node = x.node$ and $y.desc = u$. Thus, similar to the merge step in merge sort,

there must exist an iteration that performs the comparison between x and y. As a result, it is sufficient to check for update for $x.desc$ against $y.desc$ in such an iteration. □

5.3.2 Reachability Queries Using FOM

We now describe how we do the reachability query between two nodes u and v. Recall the intuitions discussed in Section 5.2.2. The following lemma formalizes this intuition:

Lemma 39. *Give two nodes u and v in G_N, we have $u \prec v$ iff one of the following is true: 1) $u \prec_d v$; or 2) $u \preceq_d w$, $w \prec v$ where w is a non-SP node (i.e., a create or put node).*

Proof. It is clear that if $u \prec_d v$, or $u \prec_d w$ and $w \prec v$, we have $u \prec v$. Now we show the other direction also holds. If $u \prec v$, there are two possibilities: 1) $u \rightsquigarrow_{sp} v$ or 2) $u \rightsquigarrow_{nsp} v$. The first case $u \rightsquigarrow_{sp} v$ is easy to see that u and v must be in the same SP-dag and thus we have $u \prec_d v$. Let's consider the second case $u \rightsquigarrow_{nsp} v$. Then the path must pass through some create or put node because all outgoing non-SP edges are incident on either create or put nodes. Now we prove that there exists a node w that $u \preceq_d w$. If u is a create or put node, then $w = u$. If not, u must have an outgoing SP edge. Therefore, we can break the non-SP path from u to v into an SP path and a non-SP path connected via some create or put node w. As a result, we have $u \rightsquigarrow_{sp} w$, i.e., $u \prec_d w$. □

Lemma 39 states that in an NSP-dag G_N, node u has a path to v iff: (1) u and v are in the same SP-dag d and there is an SP path between u and v in d; or (2) the path passes through a create or put node that breaks the path into an SP path and a non-SP path. The first case can be queried efficiently using prior work [97]. The latter case is where we apply the FOM data structures. Specifically, when querying for reachability between u and v, F-Order first queries if $u \prec_d v$ if they are in the same SP-dag; otherwise, F-Order searches whether there exists a non-SP ancestor w stored with $v.fom$ such that $u \preceq_d w$.

Algorithm 10: Group-Search in Reachability Query

```
58 Function Precedes(u, v)
59     if SP(u) = SP(v) ∧ u ≺_d v then
60         return TRUE
61     else
62         g = v.fom.find(SP(u))
63         if g then
64             return Group-Search(u, g)
65         else return FALSE
66 Function Group-Search(u, g)
67     low = 1; high = g.length
68     while low ≤ high do
69         mid = (low + high)/2; m = g[mid]
70         if u ⪯_d m.node then
71             return TRUE
72         else if m.node ≺_d u ∨ m.node ||^d_left u then
73             low = mid + 1
74         else // must be u ||^d_left m.node
75             if u ≺_d m.desc then
76                 return TRUE
77             else high = mid − 1
78     return FALSE
```

Algorithm 10 shows the pseudocode for `Precedes(u,v)` , which checks if $SP(u) = SP(v)$ and $u \prec_d v$. If so, $u \prec v$ and we are done. If not, we then check if a non-SP path exists using v's FOM data structure $v.fom$. We search for a group g with $SP(u)$ in $v.fom$; if found, we invoke `Group-Search(u, g)` to see if a w exists such that $u \preceq_d w$. If one is found, then $u \rightsquigarrow_{nsp} v$; if not, we conclude that no path exists between u and v.

By Property 2, elements in a group g is stored in a total English order. `Group-Search` uses this fact to apply a process of elimination akin to that in binary search. As hinted before, the process of elimination involves some complications — upon encountering an element x, in some cases, we must compare u against $x.desc$ (line 75) to correctly eliminate half of the remaining elements to check. This leverages Property 3 to guarantee correctness, which we discuss in Lemma 40.

Lemma 40. *Given a node u and a group g, the search in* `Group-Search(u,g)` *returns true iff there exists an element x in g such that $u \preceq_d x.node$; it returns false otherwise.*

Proof. First we show that if `Group-Search(u,g)` returns true, there exists an element x in g such that $u \preceq_d x.node$. This is evident from the code: `Group-Search(u,g)` only returns true when such a node is found (lines 71 and 76).

Now we show the other direction also hold: if an element x exists in g such that $u \preceq_d x.node$, `Group-Search(u,g)` returns true. That is, if such x exists, `Group-Search(u,g)` will find it by correctly eliminating half of the remaining elements that we don't need. We will examine this by cases on the if conditions executed in `Group-Search(u,g)`.

By Property 2, if $x.node \prec_d y.node$ or $x.node \parallel^d_{left} y.node$, then x is positioned before y in g. Let's suppose that the element we are looking for is x and it exists. If $g[mid].node \prec_d u$ (first part of condition in line 72), then obviously $g[mid].node \prec_d x.node$ and we need to only search the array elements positioned after $g[mid]$. The code correctly performs the elimination (line 73).

Now we show if $g[mid].node \parallel^d_{left} u$ (second part of condition in line 72), then we have either $g[mid].node \prec_d x.node$ or $g[mid].node \parallel^d_{left} x.node$. Consider the left subdag G_L containing $g[mid].node$ and the corresponding right subdag G_R containing u. Say G is the SP-subdag consists of G_R and G_L. Then there are two possibilities for where $x.node$ can be: either $x.node$ is in G_R, then $g[mid].node \parallel^d_{left} x.node$, or $sink(G)$ (the sink node of G) $\preceq_d x.node$, then $g[mid].node \prec_d x.node$. In either case, $g[mid]$ must be positioned before x in g, and the code correctly performs the elimination (line 73).

We now consider the case that $u \parallel^d_{left} g[mid].node$ (line 74). Again, consider the SP-subdag G with the left and right subdags G_L and G_R. Say u is in G_L and $g[mid].node$ is in G_R. Then it could be either $sink(G) \preceq_d x.node$ or $x.node \in G_L$. In the first case, $x.node$ is also a descendent of $g[mid].node$. Recall that by Property 3, $g[mid].desc$ stores the furthest descendent node of $g[mid].node$. If $g[mid].node \prec_d x.node$, we are guaranteed that $sink(G) \preceq_d g[mid].desc$. Otherwise, we have $g[mid].desc \prec_d sink(G)$ and as a result

$g[mid].desc \prec_d x.node$, which contradicts that $g[mid].desc$ is $g[mid].node$'s furthest descendent. Thus, if $sink(G) \preceq_d x.node$ is true, we must have $sink(G) \preceq_d g[mid].desc$, which leads to $u \prec_d g[mid].desc$ (line 75). Now we consider the second case, $x.node \in G_L$. In this case, we have obviously $x.node \parallel^d_{left} g[mid].node$. By Property 2, we can conclude that target x must be between positions low and $mid-1$, and the code correctly performs the elimination (line 77). □

The last part of the proof in Lemma 40 makes it clear why we must store the furthest descendent with every element — even if the target node is in the part of the array that we eliminate, we are guaranteed to find it as another element's furthest descendent field. Take nodes i and f in Figure 5.1 for instance. Say f executes second, and we want to check if f is reachable from i. We find the group g with $SP(i) = B$ and invoke `Group-Search`(i, g). Even though `Group-Search` eliminates the second half of g, it still concludes that f is reachable from i (i.e., returns true), as we have n stored as $l.desc$ and $i \prec_d n$.

By Lemmas 39 and 40 and by the operations of `Precedes`, we can show the following theorem.

Lemma 41. *Provided that $v.fom$ satisfies Properties 1–3,* `Precedes(u,v)` *correctly returns true iff $u \rightsquigarrow v$ in G_N.*

Theorem 42. *Given two executed nodes u and v in NSP-dag G_N. F-Order correctly answers the reachability query between u and v.*

Proof. By Lemmas 35, 36, 37, and 38, one can inductively show that Properties 1–3 on any FOM is maintained at all times. Then by Lemma 41, F-Order can answer the reachability query correctly provided that the FOM instance of each executed node satisfies Properties 1–3. □

5.4 The Performance Bound of F-Order

This section proves the performance bound of F-Order. To perform a reachability query between two nodes in the same SP-dag, we utilize the parallel algorithm WSP-Order [97], including scheduling support for maintaining concurrent order-maintenance (OM) data structures. In our work, we augmented our scheduler similarly to provide support for such concurrent OM data structures.

To bound the overhead of WSP-Order and the use of concurrent OM data structures, we invoke the following lemma shown by Utterback et al. [97]:

Lemma 43. *Given an SP-dag with work T_1 and span T_∞, one can perform reachability maintenance and queries on the SP-dag in time $O(T_1/P + T_\infty)$ on P processors.*

To complete the time bound for F-Order, we need to account for the additional overhead incurred by the maintenance of and queries on the FOM data structures.

Lemma 44. *Given an NSP-dag with k number of future operations, the total number of* `FOM-Insert` *invocations is at most $O(k)$, each with $O(k)$ work.*

Proof. Each FOM data structure stores only non-SP ancestors, i.e., ancestors that are either create or put nodes. Since there are k future operations, each FOM instance can have at most $O(k)$ elements. Therefore, it takes $O(k)$ time to perform a single `FOM-Insert` operation (which creates a new copy so linear work is required). Moreover, by Algorithm 7, F-Order performs `FOM-Insert` only on create and put nodes, and thus `FOM-Insert` is invoked at most $O(k)$ times. □

Lemma 45. *Given an NSP-dag with k number of future operations, the total number of* `FOM-Merge` *invocations is at most $O(k)$, each with $O(k)$ work.*

Proof. `FOM-Merge` operates on inputs of size at most $O(k)$, and thus each operation incurs at most $O(k)$ work (like the merge operation in merge sort). By Algorithm 7, F-Order

performs `FOM-Merge` only on join and sync nodes. Since there are at most k future operations, there are at most k join nodes. What's not as obvious is bounding the number of `FOM-Merge` invocations due to sync nodes.

By Algorithm 7, F-Order performs `FOM-Insert` on a sync only when the FOM data structures from both of its parents have changed from that of its corresponding spawn node.

Consider an SP-dag constructed recursively using n parallel compositions. We will call the outer-most SP-dag a level-0 dag, or G^0, and call its left and right subdags level-1 dags, G^0_L and G^0_R, or simply G^1. Since this dag is constructed with n parallel compositions, there are n levels of nested SP-dags with n sync nodes, one for each level. Without loss of generality, we will show that, by adding a new incoming or outgoing non-SP edge into this dag, the addition incurs at most one extra `FOM-Merge` on the closest enclosing sync node, but not on the other sync nodes at the outer level.

Imagine today we add an outgoing non-SP edge to the left subdag at level i, G^i_L. Consider both the sync node s that joins together G^i_L and G^i_R and its corresponding spawn node f. The FOM instance from s's left parent would change. This may or may not prompt a `FOM-Merge` at s.

Case 1: Let's consider the case where it did. Then, it must be that there is also an incoming or outgoing non-SP edge in G^i_R, causing the FOM instance from s' right parent to change from that of the $f.fom$. If so, one extra `FOM-Merge` would be incurred compared to not adding that non-SP edge. From the perspective of the sync node s' at level $i-1$, however, this change does not affect whether s' performs a `FOM-Merge` or not. Without loss of generality, say the dag consist of G^i_L and G^i_R is the left subdag at level $i-1$ (i.e., G^{i-1}_L). Without adding a new non-SP edge to G^{i-1}_R, s' will simply inherits the results of `FOM-Merge` at s.

Case 2: Let's consider the other case where this addition did not prompt s at level i to perform `FOM-Merge`. Then, it must be that, the FOM from s's right parent is the same as

$f.fom$, the FOM from the corresponding spawn node. Then, s would have simply inherit the FOM from G_L^i, incurring zero additional `FOM-Merge` operations at level i. Now, it may incur an extra `FOM-Merge` at the sync node at level j for some $j < i$. But the same argument from case 1 can be applied to level j and the `FOM-Merge` stops at level j and not further.

Thus, we can conclude that there are at most $O(k)$ total `FOM-Merge` operations, each with $O(k)$ work. □

Now we put the overhead due to maintaining FOM data structures together.

Lemma 46. *Given an NSP-dag with work T_1 and span T_∞. F-Order runs in time $O((T_1 + k^2)/P + T_\infty \lg k)$ on P processors to construct the reachability data structure, where k is the number of future operations.*

Proof. By Lemmas 44 and 45, the construction of FOM data structures incurs at most $O(k^2)$ work and in the worst case, $O(\lg k)$ multiplicative overhead on the span (if F-Order performs a `FOM-Merge` or `FOM-Insert` on every single node along the span, and one can implement `FOM-Merge` and `FOM-Insert` with additional parallelism by parallelizing merge [21] and insertion with the span $O(\lg k)$). Adding these overheads and applying Lemma 43, we obtain the bound. □

The bound shown in Lemma 46 accounts for only the construction overhead. To show the full performance bound, we must also account for the query overhead.

Lemma 47. *Given two nodes u and v in an NSP-dag G_N, a single reachability query* `Precedes(`u, v`)` *runs in time $O(\lg \hat{k})$, where $\hat{k}$ is the number of non-SP ancestors of v from the SP-dag containing u.*

Proof. In the worst case, there is no direct SP path between u and v, and F-Order needs to perform a search to check if $v.fom$ stores any descendent node of u. Identifying a group g

with $SP(u)$ in $v.fom$ takes constant time; if g exists, invoking `Group-Search(u, g)` takes at most $O(\lg \hat{k})$ time (akin to binary search). □

Theorem 48. *Given an NSP-dag G_N with work T_1 and span T_∞, F-Order can race detect G_N in parallel in time $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(\lg k + \lg r \lg \hat{k}))$ on P processors, where k is the number of future operations, r is the maximum number of readers for a single memory location, and $\hat{k}$ is the maximum number of non-SP nodes in the same SP-dag.*

Proof. The total overhead incurred due to reachability queries is related to how the access history is managed. As discussed in Section 5.2, the number of readers per memory location at a given moment can be large. However, one can still show that, the total number of reachability queries throughout the execution is bounded by $2\times$ the number of reads during execution [2]. Each query itself incurs $O(\lg \hat{k})$ overhead Lemma 47. Thus, given an NSP-dag with work T_1 and span T_∞, the total work incurred by reachability queries using F-Order is $O(T_1 \lg \hat{k})$.

Now we consider how queries impact the span of race detection. Given a single node u along the span of G_N that writes to memory location l. Since F-Order needs to perform reachability queries between u and every single reader in `reader-list`(l), assuming are at most r readers in `reader-list`(l), the total number of queries performed when executing u is at most r. Since all the queries can be computed independently of each other, such the overall query overhead has a span of $O(\lg r \lg \hat{k})$. In the worst case, every node u along the span of G_N incurs such query overhead. Then, combining Lemma 46, the bound follows. □

5.5 Implementation and Empirical Analysis

This section briefly describe our prototype implementation of F-Order and empirically evaluates its performance. We evaluate F-Order's performance across 6 different benchmarks

and compare it to FutureRD, a state-of-the-art sequential race detector for futures [98]. FutureRD provides the best sequential running time for race detecting the structured use of futures which imposes certain programming restrictions on where the future get can occur. For race detecting general futures, the algorithm by Utterback et al. [98] has an additional $\alpha(m, n)$ overhead compared to the algorithm by Agrawal et al. [2], but the algorithm by Agrawal et al. [2] has never been implemented.

FutureRD distinguishes between structured and general use of futures, and provides better running time for structured use of futures. For the purpose of comparison with FutureRD, all benchmarks are implemented with structured use of futures, and two of the benchmarks, `sw` and `hw`, have a second implementation with general use of futures. Note that, although F-Order targets programs with general futures, it works out-of-box for the structured use of futures with the same performance bound since the structured use of futures are subsumed by the general futures.

Empirical results indicate that, even though our parallel race detector incurs higher overhead on one-core execution, the overhead is never more than 3× compared to FutureRD (in fact much less for all benchmarks except for `hw`). Thus, the fact that we can race detect while executing the program in parallel quickly pays off in absolute execution times.

bench	*N*	*B*	*reads*	*writes*	*futures*	*strands*	*avg*
sw-sf	2048	64	8.59×10^9	4.20×10^6	1024	2054	1.0
hw-sf	10 (images)	-	1.73×10^{10}	1.64×10^8	3672	9914	14.05
sort	10^7	8192	2.75×10^8	2.22×10^8	14463	60030	2.95
mm	2048	64	1.72×10^{10}	1.43×10^8	18724	79577	20.94
smm	2048	64	9.40×10^8	2.50×10^5	16387	70822	8.24
ferret	simlarge	-	5.40×10^9	6.23×10^8	256	1280	13.09
sw-gf	2048	64	8.59×10^9	4.20×10^6	1024	5124	1.0
hw-gf	10 (images)	-	1.73×10^{10}	1.64×10^8	4590	11750	13.1

Table 5.1: The characteristics of the benchmarks. The `sw` and `hw` benchmarks have two implementations: structured (`sf`) and general futures (`gf`).

Implementation. We have implemented F-Order by extending the Cilk-F runtime system [86], a work-stealing runtime system that supports the use of futures. In the implementation of F-Order, we employed WSP-Order [97] for maintaining and querying the reachability between two nodes that are in the same SP-dag (i.e., $\prec_d$ and $\|^d_{left}$). We implemented the augmentation necessary in the Cilk-F runtime as proposed by Utterback et al. [97]. As discussed in Section 5.4, such an augmentation is necessary in order to provide the desired performance bound. The construction functions of the FOM are called via instrumentation inserted into the parallel control constructs of Cilk-F. The instrumentation on memory accesses are inserted via ThreadSanitizer [84] pass implemented in LLVM.

Benchmarks. Six benchmarks are used: matrix multiplication (`mm`), the Strassen matrix multiplication algorithm (`smm`), parallel merge sort (`sort`), the Heart Wall application (`hw`) from the Rodinia suite [19], Smith-Waterman for sequence alignment (`sw`), and a content-based image similarity search modified from the PARSEC benchmark suite (`ferret`) [10]. We have implemented all the benchmarks with structured use of futures. The `sw` and `hw` benchmarks, in addition, have a second implementation with a general use of futures that imposes no restrictions.

The characteristics of these benchmarks are shown in Table 5.1, including the input sizes and serial base case sizes used. We also measured the average group sizes; the measurement indicates that average group sizes (related to $\hat{k}$) are indeed small across benchmarks.

Experimental Setup. All experiments were run on a machine with two 20-core Intel Xeon Gold 6148 processors, clocked at 2.40 GHz, with hyperthreading disabled. Each core has separate private 32 KB L1 data and 32 KB L1 instruction caches, and a 1 MB private L2 cache. Each socket has a 27.5 MB shared L3 cache. The machine has 768 GB of main memory. We limit execution to the first 20 cores, located on the first socket of the machine, in order to avoid NUMA overhead. All software is compiled with LLVM/Clang 3.4.1 with

`-O3` optimizations running on Linux kernel version 4.15. Each data point is the average of 3 runs.

For each benchmark, we ran three different configurations: ***baseline***, where the benchmark is compiled without any race detection enabled; ***reachability***, where the benchmark is compiled with only the reachability component but not the access history; and ***full***, where the benchmark is compiled with full race detection.

bench	*configuration*	T_1	T_{20}	*FutureRD*
sw (structured)	baseline	21.88	2.26 [9.67×]	21.57
	reachability	22.49 (1.02×)	2.29 [9.79×]	21.52 (0.99×)
	full	697.06 (31.85×)	96.96 [7.18×]	562.55 (26.08×)
hw (structured)	baseline	15.41	0.98 [15.60×]	15.5
	reachability	17.95 (1.16×)	1.03 [17.32×]	15.5 (1.0×)
	full	943.33 (61.18×)	72.12 [13.07×]	381.85 (24.63×)
sort (structured)	baseline	1.33	0.07 [17.17×]	1.34
	reachability	5.05 (3.77×)	0.38 [13.04×]	1.34 (1.0×)
	full	30.62 (22.87×)	2.2 [13.92×]	19.48 (14.48×)
mm (structured)	baseline	8.48	0.43 [19.30×]	8.47
	reachability	12.97 (1.52×)	0.68 [19.05×]	8.47 (1.0×)
	full	484.48 (57.13×)	24.44 [19.81×]	320.46 (37.86×)
smm (structured)	baseline	2.42	0.14 [16.60×]	-
	reachability	2.84 (1.17×)	0.16 [17.30×]	-
	full	51.32 (21.14×)	2.69 [19.01×]	-
ferret (structured)	baseline	7.45	0.65 [11.41×]	7.33
	reachability	7.64 (1.02×)	0.66 [11.54×]	7.28 (0.99×)
	full	337.23 (45.26×)	32.76 [10.29×]	298.18 (40.68×)
sw (general)	baseline	21.79	2.28 [9.54×]	21.64
	reachability	24.93 (1.14×)	2.28 [10.91×]	21.58 (0.99×)
	full	704.72 (32.32×)	100.1 [7.04×]	610.75 (28.22×)
heartwall (general)	baseline	15.42	0.99 [15.48×]	15.5
	reachability	18.56 (1.2×)	1.08 [17.09×]	15.5 (1.0×)
	full	934.47 (60.6×)	68.13 [13.71×]	488.13 (31.46×)

Table 5.2: Performance of the benchmarks with F-Order and FutureRD for race detection. Execution time on P processors, T_P, is given in seconds. Numbers in the parentheses show the overhead compared to the baseline. Numbers in the brackets show the scalability relative to T_1 of the same configuration. Measurements of `smm` running with FutureRD is not available because it segfaulted.

Practical performance of F-Order

Table 5.2 shows our measurements for the benchmarks. With the exception of the `sort` benchmark, we see that the the reachability versions of F-Order incur very little overhead when compared to the baseline versions. The overhead is more pronounced in `sort` because a majority of the futures created do little more than generate more futures,[12] and even in the serial base case the work is only $\theta(B \lg B)$, where B is the problem size of the serial base case. Moreover, we expect the reachability overhead of FutureRD is less than that of F-Order on benchmarks using structured futures because its reachability algorithm designed for structured futures is more efficient than its algorithm designed for general futures. F-Order, however, cannot take advantage of the restrictions imposed by structured use of futures.

Full race detection versions incur a large increase in overhead in both F-Order and FutureRD, which comes from the combination of the memory instrumentation and the sheer quantity of reachability queries. The additional overhead of full race detection in F-Order compared to FutureRD is the price one pays to enable parallel race detection. In F-Order, each query incurs $O(\lg \hat{k})$ instead of constant time (which is the case for FutureRD). This overhead is the most evident in `hw`; this is because the structure of the parallelism and the memory access pattern cause significantly more non-SP queries than in any of our other benchmarks. Even in the case of the high overhead `hw`, however, the full race detection version of the benchmarks maintain scalability comparable to that of the baselines. The higher overhead of non-SP queries in F-Order can be offset by the scalability that F-Order gains. As shown in Table 5.2, the absolute running times of F-Order on 20 cores are significantly faster than the running time of FutureRD. In our evaluation, the running times of all benchmarks with F-Order on 4 cores or more can beat the running time of FutureRD.

[12] This is also true for `mm` and `smm`; their serial base cases, however, perform much more work, $\theta(B^3)$.

Chapter 6

Race Detection for Structured Futures

As discussed in Chapter 5, the arbitrary dependences in general futures make race detection more expensive. Due to the lack of structural properties, the reachability component in F-Order algorithm incurs $O(k^2)$ overhead for construction and $O(\lg \hat{k})$ for each query, where k is the total number of futures used in the computation, and $\hat{k}$ is the maximum number of future operations within a single "future task". And the access history must still store r accessors per memory location, leading to the overall running time of $O((T_1 \lg \hat{k} + k^2)/P + T_\infty(\lg k + \lg r \lg \hat{k}))$ for a program with work T_1 and span T_∞ running on P cores.

Interestingly, the work by Utterback et al. [98] explores a sequential algorithm for race detecting programs with ***structured futures***, which imposes certain restrictions on how futures can be used. Even though structured futures still allows for arbitrary dependences among future tasks, these programs still have more structural properties compared to general futures that allow for more efficient race detection. Utterback et al. gave a sequential algorithm for reachability analysis with an *almost* constant amortized overhead giving a total running time of approximately $O(T_1)$. However, this algorithm is inherently sequential and heavily depends on the depth-first left-to-right execution of the program.

In this chapter, we propose a parallel race detection algorithm, called ***SF-Order***, for programs with structured futures (Section 6.2). By exploiting the restrictions imposed by the structured use of futures, we are able to bring down the reachability query overhead to be constant time (although the construction overhead is still $O(k^2)$), and we are able to bound the number of readers to keep per memory location. Specifically, one can retain the same correctness guarantees while storing at most $2k$ readers per memory location. Combining these savings in overhead, our algorithm runs in time $O((T_1 + k^2)/P + T_\infty \lg k)$ on P cores, where k is the total number of futures used in the computation (Section 6.3). The interesting thing to note is that, unlike the prior results for race detecting general futures, this bound does not depend on r, the number of readers between a pair of writes. In addition, compared to the bound by prior work, this running time provides a saving of a $\lg \hat{k}$ multiplicative factor on the work term and a $\lg r \lg \hat{k}$ additive factor on the overhead on the span term.

We have implemented this algorithm in practice and empirically compared it against the state-of-the-art sequential algorithm [98] and F-Order, the parallel race detection algorithm designed for general futures (Section 6.4). Empirical results indicate that, when compared with the sequential algorithm designed for structured futures, although our algorithm has a slightly higher overhead, its absolute running time wins out when running on two cores or more. When compared to F-Order, SF-Order algorithm indeed incurs lower overhead and performs better.

6.1 Revisiting Structured Futures

As discussed in Section 4.1.2, the use of ***structured futures*** imposes the following restriction: a) ***single-touch:*** `get` is invoked on a future handle h at most once; b) ***no race on a future handle:*** there is a sequential dependence from the program point where a future handle h is created (via `create`) to the program point where a `get` is invoked on h without going through the created future task associated with h. Put it differently, given a pair of

create node to its corresponding get node (by invoking `get` on the corresponding handle), there is a directed path from the create node to the get node where the path starts from the continuation edge. Note that, given this restriction, it follows that a program that utilizes only `spawn`, `sync`, and structured futures can execute sequentially on one core (which follows the left-to-right depth-first traversal) without ever block on `sync` or `get`.

Beyond the race detection work by Utterback et al. [98], it has also been shown that the structured futures allow one to achieve better bounds on cache misses [37] and scheduling overhead [86] compared to that for general futures (also discussed in Chapter 4). Such results are interesting because the set of programs generated by structured futures is larger than the set generated by fork-join and pipeline parallelism and contains them both. Moreover, the use of structured futures is not purely of academic interests but useful in practice. The scheduling work by Singer et al. [86] show that one can implement dynamic programming applications such as Smith-Waterman sequence alignment with lower span compared to the implementation with only fork-join parallelism (albeit the improvement is constant and not asymptotic) and thereby achieve better scalability in practice. Other platforms that employ futures (e.g. [58, 85]) were also able to utilize structured futures to implement interesting application features that traditional fork-join parallelism could not achieve.

Programs use structured futures generates a class of dags that we refer to as ***SF-dags***. As we will see in Section 6.2, these dags have particular structural properties that can be exploited to perform race detection more efficiently.

Notations

An SF-dag is generated by a set of futures which can call `create` to create a new future and call `get` on future handles in a structured manner. Each future in itself is an SP-dag. Similar to NSP-dags, therefore, an SF-dag $\mathcal{D}$ can be decomposed as a set of SP-dags connected via non-SP edges. We call each individual SP-dag $F \in \mathcal{D}$ a ***future dag*** or a ***future***. We assume

that each future has a unique identifier. In addition, we say that a node $u \in F$ if u is in the SP-dag that F denotes — in this case, the instructions associated with u are part of the execution of that future. Since each future is an SP-dag, it has a unique ***first*** node which precedes all other nodes and a unique ***last*** node that all other nodes in the SP-dag precede.[13] We say that the first node of a future F is $\texttt{first}(F)$ and the last node of F is $\texttt{last}(F)$. An example SF-dag is shown in Figure 6.1.

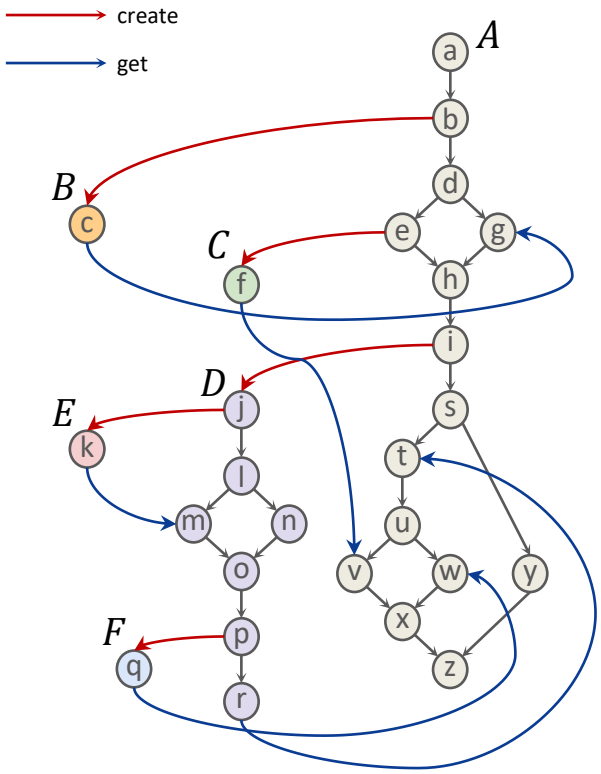


Figure 6.1: An example of an SF-dag.

We say that a future F is a ***parent*** for future G (denoted by $F = \texttt{fparent}(G)$ if some node $u \in F$ created the future G. In our example, A is the parent of B, C and D while D

[13]For future tasks, we call these nodes first and last as opposed to source and sink (for ordinary SP-dags) because a future task invoking `create` can have an escaping edge leaving the dag.

is E's and F's parent. Similarly, we say $F \in \texttt{f-ancs}(G)$ if F is either G's parent or parent of its parent and so on recursively, and $G \in \texttt{f-descs}(F)$ if $F \in \texttt{f-ancs}(G)$.

We can classify edges in three categories as we did for NSP-dags: ***create edges*** go from the strand $u \in \texttt{fparent}(G)$ that called $g = \texttt{create}(G)$ to $\texttt{first}(G)$ (all red edges in Figure 6.1); ***get edges*** go from $\texttt{last}(G)$ to the the (unique) strand that calls $\texttt{get}(g)$ where g is the future handle associated with G (all blue edges in Figure 6.1); and ***SP*** edges are all other edges. The create and get edges are also collectively called ***non-SP*** edges. Broadly, SP edges are edges between two nodes of the same future while non-SP edges are between two nodes of different futures.

Given two nodes u and v, we use $u \to v$ to denote that there is an edge from u to v and we will sometimes use subscripts such as $u \to_{get} v$ to denote that the corresponding edge is a get edge, for example. We use $u \rightsquigarrow v$ to denote the presence of a directed path from u to v. We use $u \rightsquigarrow_{sp} v$ if *at least one* path from u to v contains only SP edges and $u \rightsquigarrow_{nsp} v$ if *all* paths from u to v contain at least one non-SP edge. Note that if there are multiple paths from u to v and any one of them contains only SP edges, we say that $u \rightsquigarrow_{sp} v$.

6.2 SF-Order Algorithm

This section presents the full detail of SF-Order and its correctness proof. Recall that a race detector consists of two components — reachability analysis and access history. The access history remembers the necessary previous accessors per memory location. Upon a memory access v, the detector checks with the access history to find any conflicting previous access, say u. Then, the detector performs reachability query to see if there is a path from node u to v.

In this section, we will start by building some intuition about what data structures can help us answer the reachability queries, describe the full query algorithm, prove its correctness, and finally discuss why for race detecting structured futures, storing only $2k$

number of previous readers per memory location in access history suffice to perform race detection correctly, where k is the total number of futures used in the computation.

6.2.1 Intuition Behind the Query Algorithm

We will start by building some intuition on how the algorithm works. Recall that race detection depends on a reachability query — given two nodes u and v, we want to answer the question: is there a path from u to v. We will consider three cases:

1. If $u, v \in F$ — meaning both u and v belong to the same future dag: In this case, (as we will argue in Lemma 51) it is sufficient to check if $u \rightsquigarrow_{sp} v$ since $u \prec v$ iff $u \rightsquigarrow_{sp} v$. Note that there may also be non-SP paths between them, but at least one path will contain only SP edges. For instance, in our example dag, even though there are non-SP paths from e to u, there is also an SP path.

2. If $u \in F$ and $v \in G$ where $F \neq G$, but $F \in \texttt{f-ancs}(G)$: In this case, it is sufficient to check if there is a path from u to v that contains only create edges and SP edges. That is, (as we will argue in Lemma 53) if there is no such path, then $u \not\prec v$. Again, there may be paths from u to v that go through get edges, but at least one path will not contain any get edges. In our example, consider nodes i and q for instance.

3. If $u \in F$ and $v \in G$ where $F \notin \texttt{f-ancs}(G)$: In this case, (as we will argue in Lemma 52) it is sufficient to check if there is a path from $\texttt{last}(F)$ to v. There is a path from $\texttt{last}(F)$ to another node u iff, for all nodes $u \in F$, we have $u \prec v$.

In our query algorithm, we separately consider these three cases. For Case 1, we can rely on asymptotically optimal race detection algorithms for series-parallel dags (such as WSP-Order [97]) since we are concerned with series-parallel dependences. For Case 3, we will rely on the idea from F-Order (Chapter 5) algorithm that race detecting general futures. F-Order, for every node v, maintains a hash table that contains all the nodes u such that

$u \rightsquigarrow_{nsp} v$. However, as mentioned above, structured futures have the special property that if $u \in F$, $v \in G$, and $F \notin \texttt{f-ancs}(G)$, then $u \rightsquigarrow_{nsp} v$ iff $\texttt{last}(F) \rightsquigarrow v$. Therefore, unlike for general futures, we need not keep all such nodes u which have non-SP paths to v. Instead, for every node v, we maintain a hash table, denoted by $gp(v)$, of future IDs for all futures F where $\texttt{last}(F) \rightsquigarrow_{nsp} v$. In our example, for instance, $gp(o)$ contains B and E.

It turns out that Case 2 is the trickiest. It only applies when $F \in \texttt{f-ancs}(G)$. Therefore, for all futures G, we maintain a hash table, denoted by $cp(G)$, which contains all its ancestor futures. When checking whether $u \prec v$, we first find F and G where $u \in F$ and $v \in G$. If $F \notin cp(G)$, then this case doesn't apply. However, if $F \in cp(G)$, we have a further check. In particular, not all nodes in an ancestor future precede v — for instance, in our example, even though A is C's ancestor, $i \not\prec f$.

For this case, we will use an additional "conceptual" structure called ***pseudo-SP-dag***. A pseudo-SP-dag for an SF-dag $\mathcal{D}$, denoted by $PSP(\mathcal{D})$, is a series-parallel *approximation* of $\mathcal{D}$ which is the dag generated if we convert all `create` calls with `spawn` calls and remove all `get` calls but include an implicit `sync` at the end of a future task. Clearly, it is a series-parallel dag, since the only parallel constructs are `spawn`s and `sync`s. The pseudo-SP-dag for our example is shown in Figure 6.2. We will say $u \twoheadrightarrow v$ iff there is a path from u to v in $PSP(\mathcal{D})$. Since $PSP(\mathcal{D})$ is a series-parallel dag, we can check if there is a path from u to v in parallel using WSP-Order [97].

This $PSP(\mathcal{D})$ itself is not sufficient to check races. Pseudo-SP-dags are inaccurate for detecting races in two ways. First, they miss some paths. First, it can be the case that $u \rightsquigarrow_{nsp} v$ while $u \not\twoheadrightarrow v$; for example, even though $j \prec u$ in $\mathcal{D}$ in Figure 6.1, it is not the case in the pseudo-SP-dag in Figure 6.2. Second, and more insidiously, $PSP(\mathcal{D})$ can have ***phantom*** paths — paths that do not exist in $\mathcal{D}$. It can be the case that $u \twoheadrightarrow v$ even though $u \not\rightsquigarrow v$ in $\mathcal{D}$. For instance, in our example, $PSP(\mathcal{D})$ has a path from f to t even though such a path does not exist in $\mathcal{D}$. However, we do not use pseudo-SP-dags to check all races.

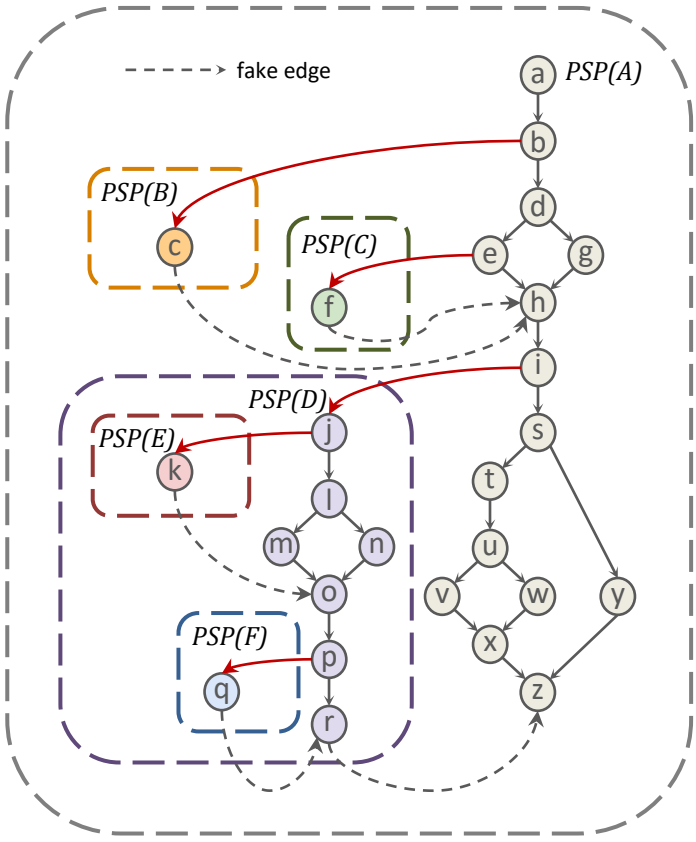


Figure 6.2: The corresponding pseudo-SP-dag for the SF-dag shown in Figure 6.1.

Recall that we only need to use $PSP(\mathcal{D})$ to check reachability from $u \in F$ to $v \in G$ if $F \in \texttt{f-ancs}(G)$. As we will argue in Lemma 57, if $F \in \texttt{f-ancs}(G)$, then for all $u \in F$ and $v \in G$, we have $u \twoheadrightarrow v$ iff $u \rightsquigarrow v$.

6.2.2 Reachability Queries in SF-Order

We can now describe the complete query algorithm. As mentioned above, in order to perform reachability queries between nodes $u \in F$ and $v \in G$, SF-Order keeps three structures.

- Order-maintenance (OM) data structures for keeping track of series-parallel relations of $PSP(\mathcal{D})$ (similar to that in WSP-Order [97]). This is used when $F \in \texttt{f-ancs}(G)$, both when $F = G$ (Case 1) and when F is a strict ancestor (Case 2). Intuitively, this data structure is used to check the existence of paths that either (1) contain

only SP edges when $F = G$; or (2) contain only create edges and SP edges when $F \in \texttt{f-ancs}(G)$.

- For each future G, $cp(G)$ is a hash table that contains the IDs of all future ancestors of G to check if $F \in \texttt{f-ancs}(G)$ so we can use $PSP(\mathcal{D})$ for Case 2.

- For each node $v \in G$, $gp(v)$ is a hash table that contains the IDs of all futures F such that $\texttt{last}(F) \rightsquigarrow_{nsp} v$ to answer queries for Case 3.

Using these data structures, the code for a query is shown in Algorithm 11. Line 2 indicates the complete query when u and v are in the same future dag. In Lemma 51 and Lemma 55 we will argue that, in this case, it is sufficient to check if there is a path from u to v in the pseudo-SP-dag. Next, Lines 4 shows the case when $F \in \texttt{f-ancs}(G)$. In this case, we check if $u \twoheadrightarrow v$ and return true if so, which is proven to be correct in Lemma 57. At this point, we have already answered the query correctly if $F \in \texttt{f-ancs}(G)$. Finally, in Lines 6, we check if $F \in gp(v)$. If so, we know that $\texttt{last}(F) \rightsquigarrow v$, which we prove in Lemma 52.

Algorithm 11: SF-Order: Reachability Query

```
1 Function Precedes(u, v)
2     if u, v ∈ F AND u ↠ v then
3         return TRUE
4     else if u ∈ F; v ∈ G AND F ∈ cp(G) AND u ↠ v then
5         return TRUE
6     else if u ∈ F; v ∈ G AND F ∈ gp(v) then
7         return TRUE
8     else
9         return FALSE
```

6.2.3 Correctness Proof of the Query Algorithm

We will now prove the correctness of this algorithm based on the intuition described above. First we start by some important structural properties of SF-dags.

Structural properties of SF-dags

We start by stating some straightforward properties of SF-dags (really for any dags with futures) — these just say that paths from one future to another must go through create and/or get edges and that the only incoming create edge is into $\texttt{first}(F)$ and the only outgoing get edge is from $\texttt{last}(F)$.

Property 1. *If $u \in F$ and $v \in G$ where F and G are distinct, then any path from u to v must contain at least one non-SP edge.*

Property 2. *Among all the nodes in F, only $\texttt{first}(F)$ has an incoming create edge (other nodes may have outgoing create edges) and only $\texttt{last}(F)$ has an outgoing get edge (other nodes may have incoming get edges).*

We now restate a couple of results shown by Utterback et al. [98]. In particular, these structural properties (unlike the ones stated above) are not true for general futures. They are properties that depend on the fact that SF-dags are generated by a structured use of futures. The following lemma is implicit in Utterback et al.'s paper [98], though not stated explicitly. In particular, in their paper, they perform race detection sequentially using a left-to-right depth-first execution and this execution satisfies the following property. Their algorithm and analysis crucially depends on this property of SF-dags.

Lemma 49. *There is some valid execution of an SF-dag such that all future descendents of F (that is, all $G \in \texttt{f-descs}(F)$) complete execution before F completes execution.*

While the model and terminology in that paper is slightly different, the following result is a straightforward restatement of Lemma 1 in Utterback et al.'s paper [98].

Lemma 50. *If $u \rightsquigarrow_{nsp} v$, then there exists at least one path from u to v that contains two sections: The first path (possibly empty) contains only get edges and SP edges and the second*

part (possibly empty) contains only create edges and SP edges. In other words there is never a create edge followed by a get edge on this path.

We will consider any such path from u to v to be a ***canonical path***. In Utterback et al. [98]'s model, there is a unique canonical path because they assume that the computation utilizes only structured futures but no `spawn`s and `sync`s. In our model, there can be many canonical paths due to the use of `spawn`s and `sync`s. For instance, in Figure 6.1, if we look at nodes c and q, there are multiple paths. There is a non-canonical path $c \rightarrow_{get} g \rightarrow h \rightarrow i \rightarrow_{create} j \rightarrow_{create} k \rightarrow_{get} m \rightarrow o \rightarrow p \rightarrow_{create} q$. However, we can choose not to go through future E and get the path $c \rightarrow_{get} g \rightarrow h \rightarrow i \rightarrow_{create} j \rightarrow l \rightarrow m \rightarrow o \rightarrow p \rightarrow_{create} q$ There is also another canonical path that goes through n instead of m.[14]

Case 1: $u, v \in F$. We first consider the (easy) case when $u, v \in F$ for some future F. The following lemma says that it is sufficient to check for SP paths. Note that this is distinct from general futures where $u \rightsquigarrow_{nsp} v$ even when u and v are in the same future.

Lemma 51. *If $u \prec v$ where $u, v \in F$, then $u \rightsquigarrow_{sp} v$.*

Proof. This property is a direct consequence of Lemma 50. Consider any path from u to v. If this path only contains SP edges, then we are done. Say this path does contain non-SP edges. Wlog, this path $\pi = u \rightsquigarrow_{sp} w \rightsquigarrow_{nsp} x \rightsquigarrow_{sp} v$ where the outgoing edge from $w \in F$ is the first non-SP edge on the path and the incoming edge to $x \in F$ is the last non-SP edge. Since $\texttt{last}(F)$ is the last node of F to execute in any execution, w is not $\texttt{last}(F)$. Therefore, the outgoing edge from w must be a create edge, since only the $\texttt{last}(F)$ has an outgoing get edge (Property 2). Similarly, x is not $\texttt{first}(F)$ — therefore, the incoming edge to F must be a get edge. Therefore, all paths from u to v that contain non-SP edges have a get edge after a create edge. Therefore, by the converse of Lemma 50, $u \rightsquigarrow_{sp} v$. □

[14] It turns out that the sequence of get and create edges on all canonical paths is the same. However, this property is not crucial for our proof.

Case 3: $u \in F$**;** $v \in G$**;** $F \notin$ **f-ancs**(G)**.** We now consider the case where $F \notin \texttt{f-ancs}(G)$ and argue that $gp(v)$ — the hash table that contains future F iff $\texttt{last}(F) \prec v$ is sufficient to check reachability in this case.

Lemma 52. *If $u \rightsquigarrow_{nsp} v$ where $u \in F$ and $v \in G$ and $F \notin \texttt{f-ancs}(G)$, then $\texttt{last}(F) \rightsquigarrow v$.*

Proof. First, we argue that if $F \notin \texttt{f-ancs}(G)$, then all paths from u to v contain at least one get edge. This is easy to see from the structure of SF-dags. SP edges only connect nodes within the same future and create edges only connect futures to descendent futures. Therefore, any path from u to v where v is not in a descendent of the future containing u must go through at least one get edge.

Now consider any canonical path p from u to v — it must contain at least one get edge. We decompose p into $u \rightsquigarrow_{sp} w \rightarrow_{nsp} x \rightsquigarrow v$ where the edge from w to x is the first non-SP edge on this path. By Lemma 50, this first non-SP edge must be a get edge. From Property 1, we know that $w \in F$ since the path from u to w only contains SP edges. Therefore, due to Property 2, $w = \texttt{last}(F)$, since $\texttt{last}(F)$ is the only node in F with an outgoing get edge. Therefore, $\texttt{last}(F) \rightsquigarrow v$. □

Therefore, when checking reachability from $u \in F$ to $v \in G$ when $F \notin \texttt{f-ancs}(G)$, it is sufficient to check if $\texttt{last}(F)$ has a path to v, which is exactly the information stored in $gp(v)$.

Case 2: $u \in F$**;** $v \in G$**;** $F \in$ **f-ancs**(G)**.** We now consider two nodes $u \in F, v \in G$ and argue the assertion stated in Case 2 — namely, if $u \rightsquigarrow v$ and $F \in \texttt{f-ancs}(G)$, there is a path from u to v which contains only create and SP edges.

Lemma 53. *If $u \rightsquigarrow_{nsp} v$ where $u \in F$ and $v \in G$ and $F \in \texttt{f-ancs}(G)$, then there is at least one path from u to v containing only create and SP edges.*

Proof. Assume, for contradiction, that when $F \in G$, there is no path from u to v containing only create and SP edges — that is, there is at least one get edge on every path.

By Lemma 50, there must be at least one path p that has all the get edges before all the create edges; and in particular, the first non-SP edge on this path must be get edge. Decompose this path into $u \rightsquigarrow_{sp} w \rightarrow_{get} x \rightsquigarrow v$. From Property 1, x is the first node on this path that is not in F and from Property 2, $w = \texttt{last}(F)$ since that is the only node in F that has an outgoing get edge.

From Lemma 49, there is some execution S where G finishes executing before F finishes execution. Therefore, there cannot be a path from $w = \texttt{last}(F)$ to $v \in G$. Hence a contradiction. □

Therefore, when $F \in \texttt{f-ancs}(G)$, we must somehow check for the existence of a path that contains only create and SP edges. This is where the pseudo-SP-dag $PSP(\mathcal{D})$ comes in. Recall that we simply convert all `creates` into `spawns`, remove all `get` statements, and include implicit `syncs` to generate $PSP(\mathcal{D})$. We say $u \twoheadrightarrow v$ if there is a path from u to v in the pseudo-SP-dag. We now argue that $PSP(\mathcal{D})$ precisely answers queries between $u \in F$ and $v \in G$ if $F \in \texttt{f-ancs}(G)$.

For convenience, we will define $PSP(F)$ for all futures F in a similar manner — the entire SP-subdag generated by F which has $\texttt{first}(F)$ as the first node and $\texttt{last}(F)$ as the last node is called $PSP(F)$. The following lemma is true due to the construction since all create edges are converted to spawn edges.

Lemma 54. *For any node $v \in G$, $v \in PSP(F)$ iff $G \in \texttt{f-descs}(F)$ (including F)*

In our example, all nodes are part of $PSP(A)$ while nodes from E and F are part of $PSP(D)$ since they are both D's descendents.

Let us consider some relationships between paths in SF-dags and the corresponding pseudo-SP-dags. The following lemma considers $u, v \in F$ and says that $PSP(F)$ precisely denotes the relationship between such nodes. This justifies our decision in Line 2 to simply check the pseudo-SP-dag when checking if $u \prec v$ when u and v are in the same future F.

Lemma 55. *If $u, v \in F$, then $u \twoheadrightarrow v$ iff $u \prec v$*

Proof. From Lemma 51, we know that if $u \prec v$ then there is an SP path between u and v. We do not remove any SP paths in the pseudo-SP-dags. Therefore, this path cannot be removed. Conversely, if there is no path between u and v, then they are in two separate SP-subdags of F. More precisely, there is node s such that u is in the left subdag of s and v is in the right subdag (or vice-versa). Therefore, in the pseudo-SP-dag, this relationship between u and v will still hold. Since pseudo-SP-dag is an SP-dag, there can be no paths between a node in the left subdag and a node in the right subdag of s just due to the properties of SP-dags. □

The next lemma states that pseudo-SP-dags are also good at finding paths that contain only create and SP edges, since the only edges removed are get edges.

Lemma 56. *If $u \in F$, $v \in G$ where $F \in \texttt{f-ancs}(G)$ and $u \prec v$, then $u \twoheadrightarrow v$.*

Proof. From Lemma 53, at least one path from u to v has no get edges and consists of only SP edges and create edges. Since the pseudo-SP-dag construction does not remove any SP or create edges, this path would still exist in $PSP(\mathcal{D})$. □

However, this in itself is not sufficient to precisely detect races since it is not obviously an if and only if statement. In particular, we might worry that $u \twoheadrightarrow v$ even if $u \not\rightsquigarrow v$. For instance, in our example, $PSP(\mathcal{D})$ has a path from f to t even though such a path does not exist in the original dag $\mathcal{D}$. These phantom paths are due to ***fake edges***, denoted by $\rightarrow_{fake}$. In particular, pseudo-SP-dags have additional sync edges that are not in the original SP-dag — these are the get edges from the last node of a child future G to a sync node in the parent future F. In our example, the offending fake edge is from f to h. We will say a path from u to v is fake (denoted by $u \twoheadrightarrow_{fake} v$) if $u \twoheadrightarrow v$, but $u \not\rightsquigarrow v$. Clearly, a fake path must have one or more fake edges.

The following lemma says that, even though there can be fake paths in pseudo-SP-dags, they do not occur between $u \in F$, $v \in G$ if $F \in \texttt{f-ancs}(G)$.

Lemma 57. *If $u \in F; v \in G$ such that $F \in \texttt{f-ancs}(G)$, then $u \twoheadrightarrow v$ implies $u \rightsquigarrow v$.*

Proof. Assume for contradiction that $u \twoheadrightarrow_{fake} v$ — that is, all paths from u to v contain at least one fake edge. Consider the path p with the smallest number of these fake edges.

Due to the way the pseudo-SP-dag is constructed, a fake edge always goes from $\texttt{last}(H)$ for some future H to some sync node in $\texttt{fparent}(H)$.

Say $X = \{F_1 = F, F_2, F_3, ..., F_k = G\}$ be the set of all the futures which are ancestors of G but not ancestors of F in order of depth in the create-tree. That is, if we look at the create-tree, these are the futures on the path from F to G.

Case 1: Some fake edge on p goes from $\texttt{last}(H)$ to some node $y \in \texttt{fparent}(H)$ where $H \notin X$. In this case, the path p can be decomposed to $u \twoheadrightarrow x \rightarrow_{create} \texttt{first}(H) \twoheadrightarrow \texttt{last}(H) \rightarrow_{fake} y \twoheadrightarrow v$ where $x, y \in \texttt{fparent}(H)$ since both $PSP(H)$ and $PSP(\texttt{fparent}(H)$ are series-parallel dags. In particular, all paths to $\texttt{last}(H)$ must go through $\texttt{first}(H)$ (unless they originate in this subdag). In addition, there must be a path directly from x to y that uses only edges within $\texttt{fparent}(H)$ since there is always a path from the create (spawn) node to the corresponding sync node. Therefore, we can replace the subpath with fake edge with a subpath without fake edge, contradicting the minimality assumption.

Case 2: All fake edges on the path p go from $\texttt{last}(F_{i+1})$ to $y \in F_i$. Therefore, there is a path $u \twoheadrightarrow \texttt{last}(F_{i+1}) \rightarrow_{fake} y \twoheadrightarrow v$. However, by Lemma 54, $G \in PSP(F_{i+1})$. Therefore, there is a path from v to $\texttt{last}(F_{i+1})$ since all nodes within a series-parallel (sub)dag have a path to the $\texttt{last}$ of that series-parallel (sub)dag. However, we cannot have $\texttt{last}(F_{i+1}) \twoheadrightarrow v \twoheadrightarrow \texttt{last}(F_{i+1})$ since $PSP(F_{i+1})$ is a dag. □

6.2.4 Maintaining the Reachability Data Structures On-the-fly

As mentioned in Section 6.2.2, SF-Order maintains three separate data structures: (1) A reachability data structure for the pseudo-SP-dag. (2) The $gp(v)$ hash table — for every node v, this table has the IDs off all futures F such that $\texttt{last}(F) \prec v$. (3) The $cp(G)$ hash table — for every future G, it stores the IDs of all future ancestors of G.

We now briefly explain how these data structures are maintained during a parallel execution. To check reachability within the pseudo-SP-dag, we use WSP-Order described by Utterback et al. [97]. The $cp(G)$ data structures is also easy to maintain. When a future G is created by future H, it simply copies over its parent's hash table ($cp(H)$) and adds its own ID to it. Maintaining $gp(v)$ is slightly more complicated, but not by much — the argument is identical to the one given in Section 5.3. Conceptually, a node simply gets the union of its parent's tables — $gp(v) = \cup_{u \to v} gp(u)$. Since we cannot afford to copy hash tables at every new node — we use pointers most of the time. If a node has a single parent, it need simply keep a pointer to its parents hash table and refer to it directly. We need only create new hash tables when a node has multiple parents and their tables must be merged — that is, at sync nodes and at get nodes. Naively merging at every `sync` and `get` is also too expensive — while there are only k get nodes in the computation (one for each future), there could be many more sync nodes. We can be cleverer about the implementation however, and only perform a merge if among the hash tables associated with the two parents of a node, each contain some item that the other does not contain. In Section 5.4, we argued that this can occur at most k times during the computation and that argument holds here as well.

6.2.5 The Access History Component

In a race detection algorithm, the access history stores the readers and writers that previously accessed a given memory location. For programs with only fork-join parallelism (i.e., SP-dags), given a memory location l, Mellor-Crummey [55] has shown that it suffices to store

one previous writer — called ***last writer***, that is simply the last writer that wrote to l, and two readers — the ***rightmost reader*** $rreader(l)$ and the ***leftmost reader*** $\texttt{lreader}(l)$. For programs with general futures, however, the race detector must store an arbitrarily large number of previous readers for each memory location [2].

By exploiting the restrictions imposed by structured futures, we show that one can store only $2k$ readers per memory location, where k is the total number of futures used in the computation, without breaking the correctness guarantees. In particular, given a memory location l and a future dag F in an SF-dag $\mathcal{D}$, SF-Order stores only the rightmost reader $rreader(l, F)$ and leftmost reader $\texttt{lreader}(l, F)$ of l with respect to F (that is, the leftmost and rightmost readers of l in F compared to all other readers of l in F). Recall from Lemma 51, if two nodes u and v are in the same future dag and $u \prec v$, then there must exist an SP path between them. Thus the following lemma straightforwardly follows from prior work by Mellor-Crummey [55].

Lemma 58. *At any point during the execution of an SF-dag, let $R_{(l,F)}$ be the set of nodes in future dag F that have read memory location l and w be any other node in F. We have $r \prec w$ for all $r \in R_{(l,F)}$ iff $rreader(l, F) \prec w$ and $\texttt{lreader}(l, F) \prec w$.*

Lemma 58 says that given a memory location l and a future F, storing its rightmost and leftmost readers suffice to detect intra-future races. Now we prove these readers also suffice to detect inter-future races.

Lemma 59. *At any point during the execution of an SF-dag, let $R_{(l,F)}$ be the set of nodes in a future dag F that have read memory location l and w be any other node in some future G distinct from F. We have $r \prec w$ for all $r \in R_{(l,F)}$ iff $rreader(l, F) \prec w$ and $\texttt{lreader}(l, F) \prec w$.*

Proof. If all $r \in R_{(l,F)}$ precede w then $rreader(l, F)$ and $\texttt{lreader}(l, F)$ must also precede w since they are in the set $R_{(l,F)}$.

Now we show the other direction — assuming that $rreader(l, F) \prec w$ and $\texttt{lreader}(l, F) \prec w$, we need to show that all $r \in R_{(l,F)}$ precede w. Since $w \notin F$, we have $rreader(l, F) \rightsquigarrow_{nsp} w$ and $\texttt{lreader}(l, F) \rightsquigarrow_{nsp} w$ (Property 1). Let's first consider the case where $F \notin \texttt{f-ancs}(G)$. Then by Lemma 52, $\texttt{last}(F) \rightsquigarrow w$, and thus all nodes in F precede w.

Next we consider the case that $F \in \texttt{f-ancs}(G)$. Since both $rreader(l, F)$ and $\texttt{lreader}(l, F)$ are in F, $w \in G$, and $F \in \texttt{f-ancs}(G)$, by Lemma 53, there is at least one path from $rreader(l, F)$ to w containing only create and SP edges. We can decompose this path into $rreader(l, F) \rightsquigarrow_{sp} x \rightarrow_{create} y \rightsquigarrow z \rightarrow_{create} \texttt{first}(G) \rightsquigarrow_{sp} w$ (where $y \rightsquigarrow z$ can be empty if F is the immediate future parent of G). Similarly with the same argument we can decompose the path from $\texttt{lreader}(l, F)$ to w into $\texttt{lreader}(l, F) \rightsquigarrow_{sp} x' \rightarrow_{create} y' \rightsquigarrow z' \rightarrow_{create} \texttt{first}(G) \rightsquigarrow_{sp} w$ (where $y' \rightsquigarrow z'$ can be empty if F is the immediate future parent of G). Since each future has exactly one parent, we have $z = z'$ and $x = x'$ inductively. Therefore, we have $rreader(l, F) \rightsquigarrow_{sp} x$ and $\texttt{lreader}(l, F) \rightsquigarrow_{sp} x$. Then based on Lemma 58, we know for any other reader r in $R_{(l,F)}$, r must precede x as well, which leads to $r \prec w$. □

6.3 Performance Analysis of SF-Order

Now we can analyze the performance bound for SF-Order. First, we can state the following bound for the reachability component based on the construction discussed in Section 6.2.4:

Lemma 60. *Given a computation with work T_1, span T_∞, and k futures, constructing the reachability data structure has total work of $O(T_1 + k^2)$ and total span of $O(T_\infty + \min\{T_\infty, k\} \lg k)$. Therefore, the running time on P processors is $((T_1 + k^2)/P + T_\infty + \min\{T_\infty, k\} \lg k)$.*

Proof. Maintaining WSP-Order to answer reachability queries between $PSP(\mathcal{D})$ has no asymptotic overhead [97]. $cp(G)$ for each future is constructed when the future is created and takes at most k extra work, for a total of k^2 overhead for k futures. As for $gp(v)$, we argued that new hash tables are created at most $O(k)$ times — once for each of the k get node and at k sync nodes at the most. Each of these merges takes $O(k)$ time since no hash table can be larger than k. Therefore, the total work is $O(T_1 + k^2)$.

Every copy and merge can be done in parallel for the span of $O(\lg k)$. Since there are at most k such merges, this overhead on the span can not be larger than $O(k \lg k)$. In addition, at most T_∞ of these merges fall along the critical path — hence the result. □

We can also show easily that queries are cheap. Utterback et al. [97] show that WSP-Order answers queries in $O(1)$ time amortized. In addition to that, we only check $gp(v)$ and $cp(G)$ once for each query, which each take $O(1)$ time in expectation.

Lemma 61. *Checking if $u \prec v$ using the query algorithm takes $O(1)$ time amortized and in expectation.*

Now we can state the final performance bound for SF-Order:

Theorem 62. *Given a computation with work T_1 and span T_∞, SF-Order executes in time $O((T_1 + k^2)/P + T_\infty \lg k)$ on P processing cores, where k is the total number of futures used in the computation.*

Proof. On a read, the the race detector has to check races against at most one previous writer and each query takes $O(1)$ time. On a write, the race detector may check races against at most $2k$ previous readers. Therefore, each write may cause up to $O(k)$ work and $O(\lg k)$ span (since all these checks can be done in parallel). However, these reads can then be removed from the access history; therefore, this $O(k)$ work can be amortized against the cost of performing them in the first place. By Lemma 60, the reachability structure construction

costs $O(k^2)$ asymptotic overhead giving us the work term. For the span, the $O(\lg k)$ overhead is multiplicative on the span; therefore, the additive overhead of construction is absorbed by it, leaving us with the total span of $O(T_\infty \lg k)$. The P processor bound follows from standard scheduling theorems. □

6.4 Implementation and Empirical Evaluation of SF-Order

We have implemented SF-Order and empirically evaluated it by comparing it against Multi-Bags [98], the state-of-the-art sequential race detector designed for structured futures, and F-Order (Chapter 5), the state-of-the-art parallel race detector designed for general futures. Experimental evaluation indicates that, although our algorithm incurs a higher overhead for one-core executions compared to MultiBags, its absolute running time wins out when running on two cores or more as MultiBags can only run the program sequentially. On the other hand, when compared F-Order, our algorithm incurs lower overheads in general, due to the lower reachability construction and query overheads.

Implementation overview

Here we briefly describe the implementation of SF-Order. As discussed in Section 6.2, the reachability component of SF-Order requires three different types of data structured. The first one is the SP-Order data structure from the WSP-Order algorithm [97] to maintain the reachability of pseudo-SP-dags. The WSP-Order algorithm requires a specialized runtime system support in order to obtain the amortized constant time query overhead on SP-Order. Such runtime system support is similarly required by F-Order, that is, an extended Cilk-F runtime system [86] that supports the use of futures and the specialized runtime system support for WSP-Order. We have taken this extended Cilk-F runtime and incorporated into our software that implements SF-Order.

For gp, recall that it is simply a hash table per node v in the SF-dag that keeps track of the IDs of all futures F such that the last node of F is an ancestor of v. One bit suffices to store such information per unique future in the execution. Thus, instead of utilizing an actual hash table hashing the unique IDs of such futures F, we utilized an array of 64-bit integers to indicate membership of $gp(v)$ — a bit in position i indicates whether the last node of a future F with ID i is an ancestor of v.

Similarly for cp, it is again a hash table containing the IDs of all future ancestors F of a given future G. Again, one bit suffices to store such information per unique future F. Thus, we similarly utilized an array of 64-bit integers to indicate membership of $cp(G)$ instead of an actual hash table.

Finally, for the access history component, we utilized the same access history construction as in F-Order — a two-level hash table that acts like a direct-mapped cache, hashing the address of a memory location to its metadata. Even though SF-Order can bound the number of readers per memory location in the access history, doing so required that we utilize yet another hash table in the metadata for a given memory location, which hashes from a future ID to its leftmost and rightmost reader. Since the overall space, hashing, and additional query overhead (to check if some reader is the leftmost or right most compared to existing readers) likely outweigh the saving in the number of readers we can omit, we simply store all the readers in the hash table between writes like what was done in F-Order.

Experimental setup

All experiments were conducted on a machine with two 20-core Intel Xeon Gold 6148 cores, clocked at 2.40 GHz. Hyperthreading and dynamic frequency scaling are disabled. Each core has a separate private L1 data cache and L1 instruction cache, with 32KB capacity each. Each core also has a 1MB private L2 cache. Each socket has a 27.5 MB L3 cache shared among 20 cores. The machine has 768 GB of main memory. We have used only one socket

for the experiments to avoid variance due to NUMA effect. All software is compiled with LLVM/Clang 3.4.1 with `-O3` optimization level, running on Linux kernel version 4.15. Each data point is the average of five runs with standard deviation less than 5.5%.

We have used five benchmarks to evaluate performance, including divide-and-conquer matrix multiplication (`mm`), parallel mergesort (`sort`), Smith-Waterman sequence alignment (`sw`), the Heart Wall application (`hw`) from the Rodinia benchmark suite [19] that tracks the movement of a mouse heart over a sequence of ultrasound images, and the Ferret application (`ferret`) adapted from the PARSEC benchmark suite [10] that implements a content-based similarity search on images. The inputs and execution characteristics of the benchmarks are shown in Table 3.1.

bench	N	B	*# reads*	*# writes*	*# queries*	*# futures*	*# nodes*
`mm`	2048	64	1.72×10^{10}	1.43×10^{8}	1.32×10^{8}	18724	79577
`sort`	10^7	8192	2.75×10^{8}	2.22×10^{8}	1.21×10^{7}	14463	60030
`sw`	2048	64	8.59×10^{9}	4.20×10^{6}	8.58×10^{9}	1024	2054
`hw`	10 (images)	-	1.73×10^{10}	1.64×10^{8}	1.75×10^{10}	3672	9914
`ferret`	simlarge	-	5.40×10^{9}	6.23×10^{8}	7.40×10^{9}	256	1280

Table 6.1: The input size (N), basecase size (B), and execution characteristics of the benchmarks, including the total numbers of reads, writes, reachability queries performed throughout the execution, the number of futures used, and the number of nodes in the computation dag.

Empirical evaluation of SF-Order

We have compared the performance of SF-Order against MultiBags and F-Order using five benchmarks described above, with the results shown in Table 6.2. Specifically, we evaluated each algorithm with two different configurations — the `reach` configuration runs the applications with only the reachability maintenance without actually performing race detections on memory accesses, and the `full` configuration that runs the full race detection. The

bench	*base* (T_1)	*base* (T_{20})	*config*	*MultiBags* (T_1)	*F-Order* (T_1)	*SF-Order* (T_1)	*F-Order* (T_{20})	*SF-Order* (T_{20})
mm	8.02	0.42 [19.10×]	*reach*	8.14 (1.01×)	11.36 (1.42×)	8.38 (1.04×)	0.64 [17.75×]	0.43 [19.49×]
			full	305.73 (37.84×)	468.59 (58.43×)	447.28 (55.77×)	23.62 [19.84×]	22.51 [19.87×]
sort	1.30	0.07[18.57×]	*reach*	1.27 (0.99×)	3.90 (3.00×)	1.35 (1.04×)	0.33 [11.82×]	0.07 [19.29×]
			full	17.56 (13.72×)	28.44 (21.88×)	26.20 (20.15×)	2.10 [13.54×]	1.86 [14.09×]
sw	20.92	2.14 [9.78×]	*reach*	20.90 (1.00×)	24.94 (1.19×)	24.25 (1.16×)	2.15 [11.60×]	2.14 [11.33×]
			full	583.78 (27.85×)	676.39 (32.33×)	555.39 (26.55×)	73.87 [9.16×]	64.75 [8.58×]
hw	14.87	0.95 [15.65×]	*reach*	14.77 (1.00×)	15.90 (1.06×)	15.22 (1.02×)	0.99 [15.91×]	0.95 [16.02×]
			full	333.35 (22.62×)	887.59 (59.69×)	676.25 (45.58×)	62.78 [14.14×]	51.77 [13.05×]
ferret	6.84	0.73 [9.73×]	*reach*	6.70 (1.01×)	7.10 (1.04×)	6.95 (1.02×)	0.75 [9.47×]	0.71 [9.79×]
			full	278.5 (42.07×)	308.14 (45.05×)	270.70 (39.58×)	29.88 [10.31×]	25.52 [10.61×]

Table 6.2: Execution times of the benchmarks shown in seconds for the baseline executions (i.e., with no race detection, shown as *base*) and when running with MultiBags, F-Order, and SF-Order for race detection with two different configurations. The first configuration shown as *reach* runs each algorithm with only the reachability construction overhead. The second configuration shown as `full` runs the full race detetion algorithm. Columns with T_1 show the execution times running on one core, and columns T_{20} show the execution times running on 20 cores. Numbers in the parentheses show the overhead compared to the baseline executions. Numbers in the brackets show the scalability relative to the T_1 time of the same configuration.

`reach` configuration incurs overhead only upon the execution of a parallel construct, and thus shows only the construction overhead for the reachability component. The `full` configuration incurs the full overhead, including the constructing the reachability, updating the access history, and performing the necessary queries into both the reachability and access history upon a memory access.

In theory , MultiBags incurs the least amount of overhead asymptotically (a multiplicative overhead in the inverse Ackermann's function, which is upper bounded by 4 for all practical purposes [23]), whereas for F-Order and SF-Order, there is an additional $O(k^2)$ overhead for the reachability construction. In practice, the reachability construction incurs rather negligible overhead for both MultiBags and SF-Order, whereas the overhead for F-Order is higher.

The reason behind SF-Order's lower overhead than F-Order in practice (despite having the same asymptotic overhead) is as follows. Like SF-Order, F-Order needs to maintain some type of hash table per node during execution (which is akin to the *gp* and *cp* data structures

needed by SF-Order). However, due to the properties of SF-dags, it suffices for SF-Order to maintain a *gp* (or a *cp*) as an array of bitmaps as opposed to using an actual hash table, whereas F-Order needs to employ a full-fledged hash table per node, which incurs higher space and time overheads. We additionally measured and compared the space overhead between F-Order and SF-Order. As shown in Table 6.3, SF-Order incurs significantly less space overhead, only 1.29% of the memory usage of F-Order on average, for five benchmarks.

bench	*F-Order*	*SF-Order*
`mm`	9.1	0.07
`sort`	7.64	0.05
`sw`	0.14	2×10^{-4}
`hw`	1.7	6×10^{-3}
`ferret`	6×10^{-3}	6.50×10^{-5}

Table 6.3: Memory usage of the benchmarks when running with F-Order and SF-Order for reachability maintenance, shown in gigabytes.

The full race detection is expensive across all algorithms. This is especially evident in the T_1 running times with the `full` configuration. Both parallel algorithms F-Order and SF-Order incur higher overheads than MultiBags, with SF-Order incurs less overhead than F-Order. This is actually in large part due to the fact that, queries into the access history needs to be synchronized with locks in the parallel algorithms. Since MultiBags executes sequentially, it does not incur such an overhead. In particular, for both F-Order and SF-Order, every time a read or a write occurs, one must acquire lock on the access history. The access history does utilize fine-grained locking (each lock represents a subset of the access history containing 16-byte memory locations), so contention is not really the issue. Rather, the high overhead stem from the sheer volume of locking operations necessary (which tracks the number of reads and writes shown in Table 6.1). Compared to F-Order, SF-Order incurs lower overhead in the `full` configuration due to its lower reachability query

overhead. In particular, SF-Order tends to have higher savings in overhead compared to F-Order on applications with large number of queries (e.g., `sw` and `hw`). However, the savings are dwarfed by the locking overhead. We have separately measured T_1 for F-Order and SF-Order without using locks in access history and confirmed that the locking overhead is indeed significant and dominates the additional overheads seem in `full`.

Even though F-Order and SF-Order incur higher overhead than MultiBags, they both exhibit scalability that closely tracks that of the baseline executions. As documented in Section 5.5 that when compared with MultiBags, F-Order wins out in absolute running times as long as four or more cores are used. Since SF-Order incurs lower overheads, when compared to MultiBags, SF-Order's absolute running time wins out when two or more cores are used.

Chapter 7

Optimizing Access Histories

The prior work on race detection has primarily focused on designing data structures and/or runtime mechanisms for maintaining the reachability component in a provably and practically efficient manner. In contrast, the access history has received little attention. Most prior race detectors maintain access history by using an optimized hashmap to maintain the mapping from each memory address to previous accesses, which allows for (amortized) constant time insertions and queries from the access history. In practice, however, the management of access history often incurs much higher overhead than the reachability component does.

To illustrate this fact, Table 7.1 shows the overhead of each component of a ***vanilla*** sequential race detector for Cilk [12, 41], a C/C++-based task parallel platform. This race detector implements SP-Order [9], a state-of-the-art algorithm which incurs constant overhead for managing reachability for fork-join parallel computations. SP-Order executes the computation sequentially; based on the parallel constructs observed during execution, it maintains a reachability data structure that can answer the queries about whether two strands are logically in parallel. The vanilla race detector uses an optimized two-level page-table-like hashmap to manage access history. In Table 7.1, the baseline column is the running time of the program without race detection. The reachability column shows the execution

	base	*vanilla reach.*		*vanilla full detection*		# *accesses* $\times 10^6$ *read*	*write*	# *intervals* $\times 10^6$ *read*	*write*
chol	0.61	0.61	(1.00×)	84.66	(139.78×)	1466.0	671.2	2.1	0.7
fft	13.55	13.59	(1.00×)	488.19	(36.03×)	2013.9	1400.9	325.4	16.3
heat	4.36	4.34	(1.00×)	367.24	(84.23×)	5274.3	1053.8	2.2	1.0
mmul	8.07	8.12	(1.00×)	355.66	(44.07×)	17712.5	536.9	33.6	8.4
sort	3.39	3.41	(1.00×)	72.27	(21.32×)	693.7	535.1	1.3	0.2
stra	1.49	1.50	(1.00×)	423.43	(284.18×)	3173.5	342.0	2.1	0.8
straz	1.54	1.54	(1.00×)	244.54	(158.79×)	3814.0	216.4	4.5	1.7

Table 7.1: Overheads of a vanilla race detector. Time shown in seconds. The first four columns from left to right show the benchmark name, its running time without race detection, that with only the reachability component, and that with the full race detection. The numbers in parenthesis show the overhead comparing to the baseline. The last four columns show the number of memory locations and intervals accessed, on the order of millions.

times that account for the compiler instrumentation and data structure updates to maintain the reachability data structure. The full column shows the execution times with both reachability and access history components and indicates access history is the most expensive component of race detection.

In this chapter, we propose mechanisms to speed up sequential race detectors for task-parallel code, focusing on optimizing the access history component. The key observation is as follows: For many task-parallel programs, a single strand typically performs many accesses to contiguous memory locations. We shall refer to a range of contiguous memory accessed by the same strand as an ***interval***. Table 7.1 shows the number of distinct (four-byte) memory words read/written and the number of intervals read/written for the tested benchmarks indicating that the number of intervals can be several magnitudes smaller than the number of memory words. If we manage access history at the granularity of intervals instead of memory words, we can reduce both the time overhead and memory footprint of the access history.

Given this observation, we propose two advances to optimize the access history. First, instead of checking races at every memory access, we wait until end of a strand and check

for races on all accesses performed by the strand at this point. This allows us to perform ***temporal*** and ***spatial*** coalescing. In temporal coalescing, we remove duplicate accesses — if the strand accesses the same memory location again and again, we only check for races and record this access once at the end of the strand, thereby reducing the number of queries to the access history and reachability data structures. In spatial coalescing, we coalesce contiguous memory accesses within a strand into intervals and invoke the access history and reachability data structures at the interval granularity.

Our race detector performs coalescing at both compile-time and runtime (Section 7.1). Some spatial coalescing occurs at compile-time when the compiler can statically detect that the memory accesses within a strand are contiguous. Doing so allows the race detector to lower the instrumentation overhead, since instrumentation (i.e., invocations to the race detector) occurs at the granularity of intervals as opposed to at every memory access. The compile-time coalescing is necessarily conservative, however, and may miss coalescing opportunities. Our detector at runtime checks for additional opportunities for coalescing. Collectively, compile-time and runtime coalescing allows us to exploit spatial and temporal locality that exist in the code to reduce both instrumentation overhead and calls to reachability and access history data structures.

The second advance is in access history data structure. Instead of storing accesses at word granularity, we store them as intervals, i.e., the start address l (inclusive) and the end address (exclusive). Doing so allows the access history to be represented in a more compact fashion, but we need a data structure that allows for efficient updates and queries of intervals. Specifically, given an interval to insert (or query), the we must find all overlapping intervals already in the data structure efficiently.

We use a balanced binary search tree data structure to maintain the access history (Section 7.2). [15] Our construction differs from normal interval trees since it enforces that no two

[15] Our implementation uses treaps [82, 92], but any balanced binary search tree would work.

intervals within the tree overlap and allows one to quickly identify all overlapping intervals. In particular, the cost of inserting and querying in our data structure for an interval x is $O(h+k)$ where h is the height of the tree and k is the number of intervals that overlap with x. By maintaining a balanced binary search tree such as a treap, our insert and query cost is bounded by $O(\lg n + k)$ (with high probability), where n is the number of intervals in the treap when x is inserted. This leads us to the overall computation time as follows: Given a computation with T_1 work — the time it takes to execute the computation on one processor — our race detector runs in $O(T_1 + n \lg n)$ time, where n is the number of intervals generated by the program. If n is small compared to T_1, which is typically the case, our race detector can race detect the computation in $O(T_1)$ time, incurring amortized constant overhead.

We have developed a race detector for task-parallel code based on this design (Section 7.3). Experiments suggest that our optimizations are beneficial. Compared to the vanilla system, which has an average overhead (geometric means) of 78.13×, our race detector incurs an average overhead (geometric means) of 18.61×, which is a 4× improvement. We also analyzed the treap operation overhead in detail, and found that a treap operation overhead tends to be dominated by the tree height as the number of overlap intervals tends to be really small. Moreover, since the treap overhead is small compared to other operations performed by the race detector, the race detector overhead remains stable as the number of intervals increases.

7.1 Compile-Time and Runtime Coalescing

This section discusses the compile-time coalescing, which can decrease instrumentation overhead, and runtime coalescing, which reduces the number of intervals and provides the additional benefit of deduplication.

7.1.1 Compile-Time Coalescing

To perform compile-time coalescing, the race detector uses the Tapir compiler [81] and leverages its representation of task parallelism. Although the details of how the compiler performs coalescing are beyond the scope of this dissertation, we examine at a high level what coalescing the compiler can and cannot do, using examples drawn from the benchmarks.

Algorithm 12: Base Case of `matmul`

Data: Submatrices A, B, and C, of size $m \times n$, $n \times p$, and $m \times p$ respectively, where each submatrix lies inside a larger $N \times N$ matrix that is stored in row-major order.
Result: $C \leftarrow C + A \cdot B$

```
1  for i ← 0 to m do
2      __coalesced_load_hook(C[i · N], p)
3      __coalesced_store_hook(C[i · N], p)
4      for j ← 0 to p do
5          t ← load(C[i · N + j])
6          __coalesced_load_hook(A[i · N], n)
7          for k ← 0 to n do
8              a ← load(A[i · N + k])
9              __load_hook(B[k · N + j])
10             b ← load(B[k · N + j])
11             t ← t + a ∗ b
12         store(C[i · N + j], t)
```

Algorithm 12 presents a pseudocode example of compile-time coalescing for the base case of the matrix-multiplication code (`mmul`) from the Cilk-5 distribution [34]. The `mmul` benchmark performs dense matrix-matrix multiplication on matrices stored in row-major order using a parallel recursive divide-and-conquer algorithm. This algorithm divides the input matrices along the longest dimension and recursively multiplies the resulting rectangular submatrices. The base case of this recursion multiplies small rectangular submatrices serially, using the pseudocode in Algorithm 12. In this pseudocode, the `load` and `store` functions denote hardware operations to load and store memory, respectively.

To instrument this code for race detection, the compiler inserts calls to the `__load_hook`, `__coalesced_load_hook`, and `__coalesced_store_hook` hook

functions to identify memory read / written. In the `__coalesced_load_hook` and `__coalesced_store_hook` functions, the first argument identifies the starting memory address loaded or stored, and the second argument specifies the amount of memory accessed. The `__coalesced_load_hook` and `__coalesced_store_hook` functions in particular identify coalesced instrumentation that the compiler inserted. For didactic simplicity, this pseudocode assumes that a single element of the matrix has size one.

As Algorithm 12 shows, the compiler is able to insert coalesced instrumentation for the loads and stores to the C and A submatrices. For the C submatrix, the compiler justifies representing accesses to C using coalesced loads and stores on lines 2 and 3 as follows. Each iteration of the j loop (lines 4–12) loads and stores memory location $C[i \cdot N + j]$. Hence, one invocation of the j loop loads and stores all of memory from $C[i \cdot N]$ up to, but not including, $C[i \cdot N + p]$. In addition, because this base case is serial, these loads and stores cannot race with any loads or stores within the same invocation of the base case. Hence, it is equivalent to represent accesses in the j loop to individual elements of C as coalesced accesses before the j loop to the memory from $C[i \cdot N]$ to $C[i \cdot N + p]$. In other words, a determinacy race will exist with a coalesced access to this range of memory addresses if and only if a determinacy race exists with a load or store to an individual element of C in the j loop. A similar analysis allows the compiler to represent the accesses to A with a coalesced-load on line 6.

Algorithm 12 also shows an existing limitation of the compiler's ability to coalesce instrumentation. In particular, line 9 shows that the compiler does not coalesce instrumentation for loads from the B matrix (line 10). In this code, the k loop (lines 7–11) reads the B submatrix in column-major order. But because the B matrix is stored in row-major order, the reads from B in the k loop do not cover contiguous memory locations. Hence, the compiler's analysis of the `load` operation on line 10 in the context of the k loop does not allow it to generate coalesced instrumentation for these loads. As a result, the compiler simply instruments the `load` on line 10 directly, using a call to `__load_hook` on line 9.

7.1.2 Runtime Coalescing

While more sophisticated compiler analysis can reveal additional opportunities to coalesce instrumentation, the inherent limitations of compile-time coalescing motivate runtime coalescing. Not only can the runtime can coalesce accesses to matrix B shown in Algorithm 12 but it can also coalesce intervals that depend on the input. For example, Algorithm 13 presents pseudocode with input-dependent memory-access patterns that are difficult to identify at compile-time. This pseudocode implements an insertion sort for the base case of the `cilksort` benchmark. In this base case, multiple executions of the inner loop (lines 4–8) may repeatedly store to the same range of memory locations between the pointers l and h. But because the store on line 6 is predicated on the comparison of input values on line 6, the compiler cannot statically determine the range of memory locations that this base case will store to. In contrast, runtime coalescing can identify these overlapping ranges and coalesce them.

Algorithm 13: Insertion-Sort Base Case of `cilksort`

```
   Data: Pointers l and h into an array A of n integers
   Result: Integers between l and h are sorted
1  q ← l + 1
2  while q ≤ h do
3  |  a ← load(q); p ← q − 1
4  |  while p ≥ l do
5  |  |  b ← load(p)
6  |  |  if b > a then store(p + 1, b)
7  |  |  else break
8  |  |  p ← p − 1
9  |  store(p + 1, a)
```

To perform runtime coalescing, we use a ***bit-hashmap*** to keep track of which memory locations are accessed during a strand's execution. The bit hashmap is a compact version of the access history hashmap used by vanilla race detector. Specifically, we use two separate

two-level page-table like hashmaps to perform runtime coalescing: one for read accesses and one for write accesses. When an access is made, the prefix and suffix of its address are used to index into the first-level and second-level tables, respectively. Tables at the second level are initialized lazily on first access. Each second-level table contains an array of 64-bit integers, where each bit represents a four-byte range. A bit is set if the corresponding word is accessed within the currently-executing strand and unset otherwise.

Runtime coalescing exploits the fact that the compiler performs some coalescing. When a coalesced load or store hook executes, the setting of the corresponding bits are done using bit tricks that employ bit-level parallelism. As the hashmap tends to be sparsely populated, vectors are used to remember indices corresponding to the first and second-level table entries set within the strand. After the strand finishes, we iterate through the stored indices to compute the intervals accessed and clear out the table entries in order to reuse the table for the next strand.

Runtime coalescing provides multiple benefits. First, as previously mentioned, runtime coalescing directly observes the program execution and can discover opportunities due to input-dependent or pointer-based operations that the compiler struggles to analyze. Second, overlapping intervals generated at two different points in the same strand are merged into a single interval. Finally, runtime coalescing provides ***deduplication***: multiple accesses to the same memory location (within a strand) are coalesced into one interval that is checked for races and inserted once into the access history. In contrast, the vanilla race detector checks for races at each access. Even though the repeated memory accesses will generate repeated updates to the runtime coalescing bit hashmaps, updates on the bit-hashmaps are significantly cheaper than those on the hashmap access history used in vanilla, because the hashmap access history keeps track of much more data in order to perform race detection. As we shall see in Section 7.3, both compile-time and runtime coalescing provide benefit, but the runtime coalescing provides greater benefit due to these reasons.

7.2 Interval-Based Access History

We now describe the access history data structure that efficiently supports (1) query to find all intervals that potentially conflict with a given interval; and (2) update to the data structure to insert the new interval. We also analyze the theoretical performance of this data structure.

Recall that in a sequential race detector for fork-join parallelism, it suffices for each memory location to store its last writer and left-most reader [27]. In a traditional access history data structure, when a strand s writes to this memory location ℓ, we check if s is in parallel with the left-most reader of ℓ or with the last writer and declare a race if so. The strand s is now stored as the last writer of this location. Similarly, if s reads this memory location, we check if s is in parallel with the last writer and declare a race if so. We then check if s is left-of the existing left-most reader and store s as the left-most reader if so.

We want to store intervals instead of individual memory locations in the access history. We keep separate data structures for read intervals and write intervals. Each of these will store interval objects, say x with three fields: $x.start$ and $x.end$ denote the beginning (inclusive) and end (exclusive) of the interval and $x.accessor$ stores the strand we want to store — the last writer for the write data structure and the left-most reader for the read data structure. When convenient, we denote an interval as three-tuple: $[\texttt{start}, \texttt{end}, \texttt{accessor}]$. The intervals stored within each data structure must be disjoint from each other since each memory location can have at most one last writer and one left-most reader.

When a new strand s generates a read or a write interval, it is also represented as an interval object o with the appropriate *start* and *end* values and accessor s. We must check if any access within o races with any pre-existing access within the access history and if so, report a race and then update the access history. However, this is not

straightforward. Consider the following example. Say we had the following read intervals: $[8, 16, a], [24, 32, b], [40, 52, c], [52, 60, d]$. We get a new read interval $[12, 56, e]$. The tree after the update depends on the relationship of e with all other intervals. Say e is left of a and c, but not b and d. After the update, the data structure must store $[8, 12, a], [12, 24, e], [24, 32, b], [32, 52, e], [52, 60, d]$. Therefore, a new interval may overlap with many previous intervals and some previous overlapping intervals may remain entirely, and some may be removed or trimmed.

Intervals are stored in two binary search trees (one each for read and write intervals) keyed by the *start* field of the interval. The data structure is similar to interval trees [23][Chp.14.3]; however, we enforce the additional ***non-overlapping*** property that all intervals in the tree must be disjoint. The two trees behave a little differently since the accessor for each interval must be the last writer in the write tree and the left-most reader in the read tree.

Say we are processing strand x with accessor s. Since the strands are processed by the race detector in sequential order, all previous intervals already in the tree are "before" s in sequential order. Therefore, if x overlaps any pre-existing interval in the write tree, then x is kept since s is always the last writer and the old interval is trimmed or removed. As we saw in the example above, this is not true in the read tree since s may not be the left-most reader for all memory locations in x. Therefore, when we see an overlap, we must check whether the old reader or the new reader is the left-most reader. We will first describe how we insert an interval in the write tree and then the read tree. We then describe how we do queries.

7.2.1 Updating the Write Tree

Given a tree T (as a pointer to the root) and a write interval x, we will use a recursive procedure to update the tree to reflect the accesses represented by interval x. We will

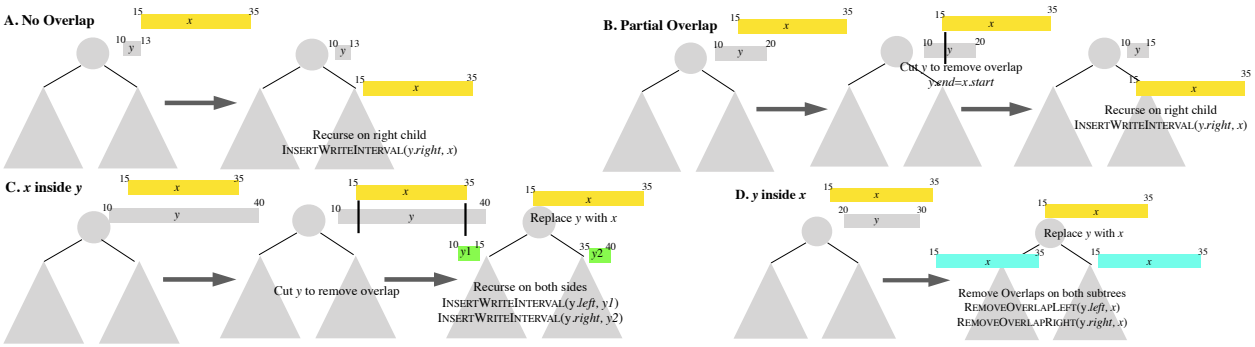


Figure 7.1: All cases illustrating INSERTWRITEINTERVAL(y, x) — assumes and maintains the no-overlap invariant.

remove/trim all intervals that overlap with x to maintain the invariant that no intervals overlap with each other in the tree.

The procedure is illustrated in Figure 7.1. There are two main procedures, INSERTWRITEINTERVAL and REMOVEOVERLAP. The INSERTWRITEINTERVAL is the main procedure that is called at the root of the tree. The procedure is called recursively as we walk down the tree. When we are at a particular tree node, say y in the tree and trying to insert node x, we can be in the following 4 cases.

A. **No overlap:** As shown in Figure 7.1(A), when y and x don't overlap, we simply recurse down to one of its children. In particular, if x is completely to the right of y ($y.end \leq x.start$), no intervals in y's left subtree can overlap with x since they all end before $y.start$. However, some intervals in y's right subtree that overlap with x, so we recurse by calling INSERTWRITEINTERVAL$(y.right, x)$. If $y.right$ is empty, then we simply insert this interval at this leaf.

B. **Partial overlap:** Figure 7.1(B) illustrates the operations when x partially overlaps with y and is to the right of y ($y.start < x.start$, $x.start < y.end < x.end$). In this case, $y.accessor$ is the last writer for part of the old interval (from $y.start$ to $x.start$), but $x.accessor$ is the last writer for memory locations after it. Therefore, we set $y.end = x.start$. Again, no intervals in the left subtree of y can intersect with x, and we recurse on the right subtree

of y by calling InsertWriteInterval($x, y.left$). A symmetric procedure is used when x is to the left of y for both this case and the previous case.

C. **Full overlap; old interval y bigger:** Figure 7.1(B) illustrates the operation when y fully encompasses x ($y.start \leq x.start$ and $y.end \geq x.end$). We have up to three intervals $[y.start, x.start, y.accessor]$, $[x.start, x.end, x.accessor]$, $[x.end, y.end, y.accessor]$.[16] We keep any one of these intervals at this location in the tree (replacing the old y) and recurse down the tree to insert the other two intervals. In Figure 7.1 we keep the middle interval and insert the left and right intervals. Note that none of these intervals overlap with any other interval in the tree since they collectively made up y which was already in the tree before and didn't overlap with anything. Therefore, we will fall into case A from now on out — we just walk down the tree and insert in the appropriate leaf.

D. **Full overlap; new interval x bigger:** Figure 7.1(B) illustrates case where x fully encompasses y. Now y can be removed from the tree entirely and replaced with x. However, there may be more intervals in both the left and right subtrees of y which also overlap with x. Therefore, we use a function called RemoveOverlap to find and remove/trim these intervals. There are two versions of this function: RemoveOverlapLeft(y, x) which is called on the left subtree and RemoveOverlapRight(y, x) which is called on the right subtree. This function is illustrated separately in Figure 7.2 and explained below.[17]

We now describe RemoveOverlapLeft(T, x). (RemoveOverlapRight(T, x) is symmetric.) Recall that RemoveOverlapLeft(z, x) is first called when a newly inserted interval x replaced an interval y which was fully within x and z was the left child of y. The general invariant is that RemoveOverlapLeft(z, x) is called on a node z when x has been inserted into some ancestor of z to z's right (therefore, $x.end \geq z.end$) and the purpose

[16] There may be fewer than 3 intervals if one or both end points are equal for x and y; this is easily handled as a special case.

[17] We can (optionally) trim x so that it ends at $y.start$ when calling RemoveOverlap. The explanation is easier without, however.

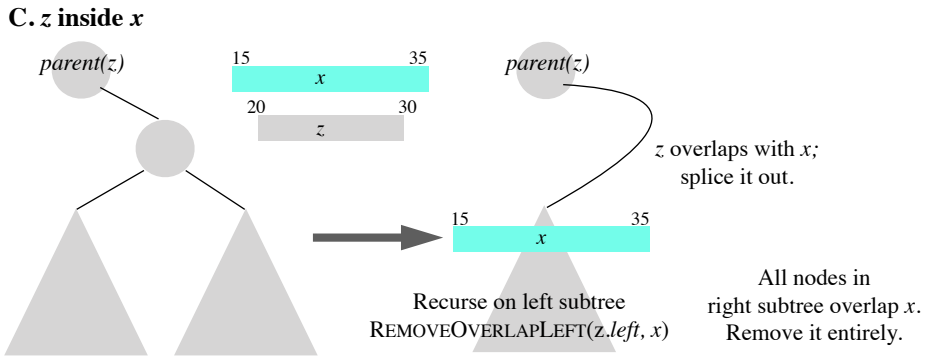


Figure 7.2: Case C of REMOVEOVERLAPLEFT(z, x). x was inserted at an ancestor to the right of z.

is to find and remove/trim intervals that overlap with x. This is also a recursive function and a subset of its cases are illustrated in Figure 7.2. Note that there are fewer cases since z cannot fully encompass x due to the invariant of this function. We also show $parent(z)$ (the old y in the example used in INSERTWRITEINTERVAL) in these figures for two reasons. First, as we will see soon, we need it for one of the cases. More importantly, we wanted to point out that even though REMOVEOVERLAPLEFT is initially called on the left child, as we make recursive calls, it can be eventually called on a right child — the function remains unchanged regardless.

A. **No overlap:** If x and z don't overlap with x to the right of z ($z.end < x.start$; x can not be to the left of z for REMOVEOVERLAPLEFT due to the invariant stated above), there can be no overlap in the left subtree of z. Therefore, we recurse on the right subtree by calling REMOVEOVERLAPLEFT$(z.right, x)$.

B. **Partial overlap:** If x and z partially overlap ($z.start < x.start < z.end$), we trim the interval z by setting $z.end = x.start$. Again, the left subtree of z cannot overlap with x. Since x is some ancestor of z to the right, the entire right subtree of z must now overlap with x and can be removed, thereby terminating the recursion.

C. **Full overlap** Figure 7.2(C) illustrates the case where x fully encompasses z. Again, the entire right subtree of z must overlap with x and is removed. In addition z itself is spliced out and replaced with its left subtree by changing the child pointer for $parent(z)$ and the parent pointer of $z.left$. In addition, we recurse on the left subtree of z by calling RemoveOverlapLeft$(z.left, x)$ to find any additional intervals that may intersect with x.

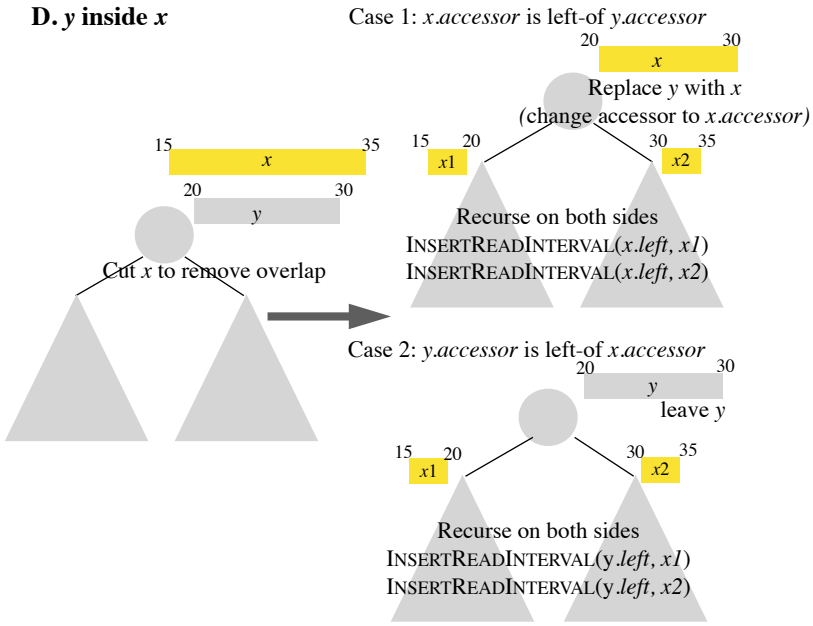


Figure 7.3: Case D of InsertReadInterval(y, x).

7.2.2 Inserting an Interval in the Read Tree

Maintaining the read tree is more complicated. In a read tree, when the new interval x overlaps with some old interval y, we may keep the y if it is left-of x. As seen in the example at the beginning of the section, this can lead to some intervals being removed and trimmed while the new interval may also be trimmed in many pieces. We have similar cases as the

write tree, but the cases are handled differently. As with the write tree, we only show one direction — where x is to the right of y — the other case is symmetric.

A. **No overlap:** This case is identical to the write tree — we simply recurse to the appropriate subtree.

B. **Partial overlap:** The case of partial overlap is slightly more complicated. We have two cases. If the new accessor $x.accessor$ is left-of the old accessor $y.accessor$, then the accessor for $[y.start, x.start]$ is old $y.accessor$ but the accessor for $x.start$ onwards is the new $x.accessor$. Therefore, this case is handled like the write tree — we cut the old interval y down by setting $y.end = x.start$ and recurse to the right subtree by calling InsertReadInterval$(y.right, x)$. If, on the other hand, the old interval's accessor $y.accessor$ is left-of $x.accessor$, then we must keep the entire interval y intact. In this case, we trim x by setting $x.start = y.end$ and use this modified interval x to recurse to the right subtree.

C. **Full overlap; old interval y bigger:** This is the easiest case. If the new interval is left of the old interval, then the read tree behaves like the write tree — the old interval is cut into three portions, one of the portions is kept at this location, and the other two are inserted with guaranteed no further overlap. If the old interval is left of the new interval, we just keep the old interval; nothing changes and we are done.

D. **Full overlap; new interval x bigger:** As illustrated in Figure 7.3(D), the case where x fully encompasses y is the most different from the write tree since we cannot simply remove y. First, y might be left of x and therefore must be kept. Even more importantly, there may be other intervals within y's subtrees that overlap with x and have accessors left of x. Therefore, we cut x into three pieces. The middle portion stays here and is labeled with $x.accessor$, if $x.accessor$ is left-of $y.accessor$, or $y.accessor$ otherwise. The other two portions are inserted into the left and right subtrees by recursing.

7.2.3 Queries to Check for Races

In order to check for races with interval x, we must find intervals that overlap with x. Note that the procedure InsertWriteIntervals already finds all overlapping intervals as it walks down the tree. [18] The main wrinkle is that when we are in case B or C of RemoveOverlap and remove entire subtrees, we must walk through those subtrees to check for races with all intervals in that subtree. In summary, the race detection procedure works as follows. For a write interval x, first check for races in the read tree by using a procedure similar to InsertWriteInterval, but making no modifications to the tree itself. Then insert into the write tree while checking for races as we go. For a read interval x, first check for races in the write tree by using a procedure similar to InsertWriteInterval, but making no modifications to the tree. Then insert into the read tree by using InsertReadInterval.

7.2.4 Performance Analysis

We have described the algorithm with a generic binary search tree. In order to keep the height low, we use a balanced binary search tree such as a treap which has height $O(\lg m)$ with high probability if there are m nodes in the treap.

We first bound the number of intervals that can be in the data structure at any given time.

Lemma 63. *When* InsertWriteInterval *(resp.* InsertReadInterval*) has been called on the root of the write tree (resp. read tree) m times, then the total number of intervals in the respective tree is $O(m)$.*

Proof. We first look at the easier case of the write tree. First, note that RemoveOverlap doesn't add any new intervals, only removes or trims existing ones and neither does case A for InsertWriteInterval. Cases B and D also just trim intervals, but do not add new

[18] In fact, InsertWriteInterval also finds all overlaps — InsertWriteInterval is more efficient, however.

ones. The only way a new interval is added is (a) x reaches a leaf node in case A and gets added (adding only one interval); or (b) in case C, we split an existing interval y and insert y_1 and y_2. In this case y_1 and y_2 are guaranteed to have no overlaps and get added at the leaves, causing an additional two intervals. Therefore, every time we insert a new interval, we add at most two additional intervals to the tree.

Now consider the more complicated case of the read tree. Again, just like the write tree, cases A–C do not add intervals to the tree. However, case D is interesting, because we call InsertReadInterval on both subtrees. There is no guarantee that these new x_1 and x_2 won't also overlap with additional intervals further down the tree and subdivide further. In the worst case, if we had i intervals before a particular interval was added, we can have $2i+1$ intervals after it was added. Consider the following example: say we had $[1, 2, a]$, $[3, 4, b]$, and $[5, 6, c]$ in the tree. If we read an interval $[0, 7, d]$ where a, b, c are all left-of d, our tree will contain $[0, 1, d]$, $[1, 2, a]$, $[2.3.d]$, $[3, 4, b]$, $[4, 5, d]$, $[5, 6, c]$, and $[6, 7, d]$.

However, it turns out that the total number of intervals cannot double with every insertion. We will see this by counting not just intervals, but also ***gaps***. Gaps are memory ranges between consecutive intervals — in our example before d is inserted, $[0, 1]$, $[2, 3]$, $[4, 5]$, $[5, -]$ are gaps. When we insert an interval that doesn't overlap with any existing interval, we increase the number of intervals by exactly one and we increase the number of gaps by at most one, for a collective increase of at most 2 (we may not increase the number of gaps if the new interval is right next to another or decrease the number of gaps by one if we fill in the gap between two intervals). When we insert an interval that overlaps other intervals, we may increase the number of intervals by a lot, but only by filling in gaps. Therefore, the collective increase in the number of gaps and intervals is at most two in all cases. An empty tree has one gap. Since each insert increases the number of gaps and intervals (collectively) by at most 2, the total number of intervals is at most $2m + 1$. □

Lemma 64. *Inserting an interval and querying into the access history takes $O(h + k)$ time where h is the height of the larger tree (read or write) and k is the number of intervals that overlap with x across both trees.*

Proof. Let us first consider the InsertWriteInterval(T, x). At every node, this recursive procedure is called on at most one child. In addition, only case A can be called multiple times; cases B–D occur at most once for every insertion. This is particularly important for case D since it calls RemoveOverlap on both children, but since this can happen at most once, the total number of recursive calls remains $O(h)$. In addition RemoveOverlap is also called on at most one child at every level. Therefore, the total running time of this procedure is $O(h)$.

We might be tempted to say that queries also take $O(h)$ since we use a similar procedure as InsertWriteInterval. However, this would be impossible — if k intervals overlap with x, then we can not possibly check for races in time less than $O(k)$. However, recall that during RemoveOverlap case B and C, when we remove entire subtrees, during queries, we must check all the nodes in these subtrees for races. Since we check precisely the set of intervals that overlap with x, the running time is $O(h + k)$.

Now, let us consider the more complicated case of InsertReadInterval. In particular, note case D; here, we call the function recursively on both children. In addition, this can happen many times (every time x overlaps fully with y — at most k times). Therefore, naively, we might conclude that the running time is $O(kh)$.

However, we can do a more careful analysis. Consider that case D happened when inserting node x on some node y and then again at some descendent node z where z is to the left of y. In this case, the right subtree of z must entirely overlap with x and we can charge the recursion on this right subtree of z to k. Therefore, apart from the first time in the tree when we fall into case D, every other time we fall into this case, we can charge one of the two recursive calls to k. Pictorially, when we insert x into the read tree, we walk at

most two root to leaf path in addition to walking to all the intervals that overlap with x for a total running time of $O(h + k)$. □

Theorem 65. *Across the entire computation, the total cost of checking for races is* $O(n \lg n + T_1)$ *where* n *is the total number of intervals generated by the program and* T_1 *is the work.*

Proof. The total number of intervals in either tree never exceeds $O(n)$ from Lemma 63. Therefore, from Lemma 64, the cost of each individual interval is $O(\lg n + k)$ if we use a balanced tree. If an interval overlaps k other intervals, then it must have size at least k and therefore, the program must do k work to generate this interval. Therefore, over all intervals, the total cost of race detection is $O(n \lg n + T_1)$ where $n \lg n$ term comes from adding the $\lg n$ cost over n intervals and T_1 comes from adding k over all intervals. □

7.3 Empirical Evaluation

In this section, we empirically evaluate our detector and the impact of the optimizations described in Sections 7.1 and 7.2. Experiments suggest that our optimizations are beneficial. Compared to the vanilla system, which has an average overhead (geometric means) of 78.13×, our race detector incurs an average overhead (geometric mean) of 18.61×. Detailed analysis indicates that the overhead of a single treap operation is dominated by the tree traversal as the number of overlapping intervals tend to be small. Moreover, since the treap overhead is small compared to other operations performed by the race detector, the race detector overhead stays constant as the number of intervals increases.

Experimental setup

We used seven standard task-parallel benchmarks (where b indicates base-case size and other parameters describe the input size): Cholesky decomposition (`chol`, $n = 2000, z = 20000, b = 16$); parallel mergesort (`sort`, $n = 2.5e^7, b = 2048$); fast-Fourier transform (`fft`, $n = 2^{26}, b = 128$); heat diffusion simulation on a 2D grid (`heat`, $nx = 2048, ny = 2048, b =$

10); matrix multiplication (`mmul`, $n = 2048, b = 64$), and two versions of Strassen's algorithm for matrix multiplication, `stra` and `straz`, which use row-major order layout and Morton Z layout, respectively ($n = 2048, b = 64$).

All experiments were run on a machine with two 20-core Intel Xeon Gold 6148 processors, clocked at 2.40 GHz, with hyperthreading disabled. Each core has separate private 32 KB L1 data and 32 KB L1 instruction caches, and a 1 MB private L2 cache. Each socket has a 27.5 MB shared L3 cache. The machine has 768 GB of main memory. All software is compiled with the Open Cilk [80] compiler originally based on Tapir [81] with `-O3` optimizations running on Linux kernel version 4.15. We have modified the compiler to perform compile-time coalescing as discussed in Section 7.1. Each data point is the average of 5 runs with standard deviation $< 4\%$.

Overview of results

We ran the benchmarks with four versions of the race detector to tease out the impact of each optimization:

- ***vanilla*** that employs an optimized two-level page-table like hashmap to manage access history and uses a compiler that generates instrumentation for each memory access;
- ***compiler*** that introduces the compile-time coalescing discussed in Section 7.1.1, with the same hashmap to manage access history as in vanilla;
- ***comp+rts*** that includes both compile-time and runtime coalescing discussed in Section 7.1 but still uses the same hashmap to manage access history; and
- ***treap*** that includes both coalescing and uses the treap construction in Section 7.2 to manage access history.

All versions utilize the same implementation based on the SP-Order algorithm [9] to maintain reachability.

These different versions allow as to gauge the impact of each optimization. By comparing vanilla and compiler, we can gauge how much instrumentation overhead is reduced. By comparing compiler and comp+rts, we can gauge how much overhead the full coalescing (i.e., coalesce as much as possible) reduces. By comparing comp+rts and treap, we measure the impact of using a treap instead of a hashmap, which incurs higher overhead per operation but reaps the full benefit of coalescing.

	vanilla		*compiler*		*comp+rts*		*treap*	
`chol`	84.66	(138.79×)	82.87	(135.85×)	26.73	(43.82×)	19.22	(31.73×)
`fft`	488.19	(36.03×)	368.76	(27.21×)	304.92	(22.50×)	489.71	(36.14×)
`heat`	367.24	(84.23×)	326.03	(74.78×)	144.43	(33.13×)	23.24	(5.32×)
`mmul`	355.66	(44.07×)	345.08	(42.76×)	219.25	(27.16×)	220.82	(27.36×)
`sort`	72.27	(21.32×)	69.39	(20.47×)	40.63	(11.98×)	15.81	(4.66×)
`stra`	423.43	(284.18×)	414.52	(278.20×)	96.30	(64.63×)	38.33	(25.74×)
`straz`	244.54	(158.79×)	244.36	(158.68×)	100.15	(65.03×)	51.66	(33.62×)

Table 7.2: Execution times (in seconds) and overheads of different versions of the race detector compared to the baseline (i.e., no race detection), whose values are shown in Table 7.1.

Table 7.2 shows the race detection overhead compared to the ***baseline*** execution time, i.e., no race detection, running each version of the detector. For most benchmarks, each additional optimization brings some benefit to the overhead reduction, leading to the final result, where treap incurs on average (geometric mean) of 18.61× overhead, much less than that of vanilla, 78.13×. The only exception is `fft`, whose overhead increases from comp+rts to treap; we explain the reason for this overhead increase as we analyze more empirical data later in the section.

Compile-time vs. runtime coalescing

Now we analyze in more detail the benefit of compile-time versus runtime coalescing. The overhead decreased between vanilla and compiler but not as substantially as between vanilla and comp+rts for the following reasons. First, comp+rts is able to coalesce more. Although

	vanilla		*compiler*		*both*		*compiler*		*both*		*compiler*		*both*	
	acc. (r)	*acc. (w)*	*int. (r)*	*int. (w)*	*int. (r)*	*int. (w)*	*avg. (r)*	*avg. (w)*	*avg. (r)*	*avg. (w)*	*sum (r)*	*sum (w)*	*sum (r)*	*sum (w)*
`chol`	1466.0	671.2	1430.3	100.6	2.1	0.7	8.44	105.3	977.2	873.6	11510.2	10103.6	1914.5	641.9
`fft`	2013.9	1400.9	2013.8	1007.6	325.4	16.3	4.9	7.5	29.28	462.45	9474.3	7168.0	9084.6	7168.0
`heat`	5274.3	1053.8	43.2	1053.8	2.2	1.0	1946.4	15.95	9635.3	16375.9	80137.7	16032.0	20004.9	16032.0
`mmul`	17712.5	536.9	17196.5	16.8	33.6	8.4	4.1	128.0	128.0	128.0	67568.0	2048.0	4096.0	1024.0
`sort`	692.7	535.1	297.8	535.1	1.3	0.2	18.6	8.0	2256.7	13042.0	5286.1	4083.4	2870.4	2861.0
`stra`	3173.5	342.0	2665.7	342.0	2.1	0.8	16.7	12.2	1886.8	2926.6	43244.5	3967.1	3824.4	2312.7
`straz`	3814.0	216.4	3814.0	216.4	4.5	1.7	14.5	16.0	2048.0	2048.0	52708.5	3302.0	8804.5	3302.0

Table 7.3: Various execution statistics on memory accesses generated when running *vanilla*, with comiler coalescing (*compiler*), and with both compiler and runtime coalescing (*both*). The *acc.* and *int.* indicate the number of accesses / intervals that eventually made into the access history, shown in millions. The *avg.* shows the average size per interval accessed, and the *sum* shows the total size (in Mbytes) accessed. The *(r)* / *(w)* indicate read or write.

the access history in both comp+rts and compiler handles a given interval at four-byte granularity, the number of intervals generated is correlated with the number of top-level calls into the access history. Thus, comp+rts incurs less function-call overhead to query and update the access history. Second, comp+rts takes advantage of the runtime deduplication, which results in fewer updates to the hashmap.

To get a better sense of how much the compile-time versus runtime coalescing can do, we separately collected various memory access pattern generated by running vanilla, compiler (compile-time coalescing) and by comp+rts (both compile-time and runtime), shown in Table 7.3. First, we shall examine the numbers shown on the left side of the table: the numbers of accesses / intervals generated by all three version. In some benchmarks, such as `mmul` and `heat`, the compiler was able to coalesce in a non-negligible way, but in most benchmarks, the compiler cannot coalesce as much. The runtime coalescing on the other hand, seems much more effective in coalescing and deduplicating, leading to two or three order of magnitude of decrease in the number of intervals. Moreover, the average sizes of intervals (*avg.*) tend to be a lot larger with runtime coalescing.

Another question is, how much role does the runtime deduplication (removing duplicate accesses) take in reducing the overhead. We can gauge the answer to this question by looking at the total bytes that made into the access history (*sum*), also shown in the table. If the

runtime performs coalescing only but not deduplication, the total bytes accessed should not change from compiler to comp+rts. Thus, by comparing the total bytes accessed generated by compiler versus comp+rts, we can tell that most benchmarks benefit from deduplication.

This data, combining with the data in Table 7.2, indicate that while both compile-time and runtime coalescing can be beneficial, the benefit from runtime is more significant.

Hashmap vs. treap

	comp+rts	*treap*
`chol`	8.93	1.41
`fft`	207.72	392.50
`heat`	123.63	2.43
`mmul`	15.94	17.51
`sort`	26.36	1.54
`stra`	59.60	1.62
`straz`	52.00	3.50

Table 7.4: The total time (in seconds) each benchmark spent updating its access history (i.e., hashmap for comp+rts and treap for treap).

Now we analyze the overhead of the treap construction in more detail and also explain why `fft` sees an overhead increase from comp+rts to treap. We measured the time that comp+rts and treap spent updating their respective access histories. Indeed, the treap overhead is much larger than that of the hashmap in `fft`. Table 7.4 shows the results.

There are multiple factors at play here. Given an interval of size x, the hashmap needs to perform $2x$ operations (insert and query). On the other hand, while the treap can reap the benefit of coalescing fully, an update to the treap incurs $O(h + k)$ time, where h is the height of the treap and k is the number of overlaps.

It turns out that while coalescing reduces the number of intervals, the reduction for `fft`, compared to other benchmarks, is less significant, and the resulting interval size is smaller as well Thus, the trade-offs made by using a treap do not work well for benchmarks with

characteristics like `fft` (i.e., less reduction in the number of intervals and smaller interval size).

Analysis of treap overhead

	input	*base*	*comp+rts*	*treap*	*hash oh*	*treap oh*	*hash ops*	*treap ops*	*# nodes*	*# overlaps*
`fft`	2^{24}	2.33	58.65 (25.17×)	80.21 (34.42×)	41.45	63.01	$2.60e^8$	$1.42e^8$	29.29	0.97
	2^{25}	5.36	125.66 (23.44×)	180.03 (33.59×)	88.44	142.81	$5.21e^8$	$2.85e^8$	28.54	0.97
	2^{26}	13.55	304.92 (22.50×)	489.71 (36.14×)	207.72	392.50	$1.22e^9$	$6.83e^8$	29.56	0.98
`mmul`	1024	1.01	27.20 (26.93×)	27.03 (26.76×)	1.82	1.65	$4.19e^7$	$1.05e^7$	16.50	0.69
	2048	8.07	219.25 (27.16×)	220.82 (27.36×)	15.94	17.51	$3.36e^8$	$8.39e^7$	19.31	0.69
	4096	65.98	1763.03 (26.72×)	1793.49 (27.18×)	122.05	152.51	$2.68e^9$	$6.71e^8$	21.54	0.70
`sort`	$5.0e^7$	7.17	88.68 (12.37×)	34.32 (4.79×)	57.99	3.63	$8.53e^8$	$7.02e^6$	36.53	1.88
	$1.0e^8$	14.99	179.12 (11.95×)	70.80 (4.72×)	115.72	7.40	$1.71e^9$	$1.45e^7$	38.67	1.90
	$2.0e^8$	31.57	389.38 (12.33×)	152.76 (4.84×)	254.27	17.65	$3.81e^9$	$3.21e^7$	40.42	1.90

Table 7.5: Execution times of `fft`, `mmul`, and `sort` running on baseline (*base*, i.e., no race detection), comp+rts, and treap on different input sizes, with overhead compared to *base* shown in parenthesis. On the right of the execution times, we also show various stats for comp+rts (using a hashmap) and treap, where the *oh* indicates time spent on access history only, the *ops* indicates the number of hashmap / treap operations, the *# nodes* shows the average number of nodes visited per treap operation, and the *# overlaps* shows the average number of overlaps encountered per treap operation.

To better understand the treap operation overhead, we collected more data using three representative benchmarks, `fft`, `mmul`, and `sort`, where the treap version performs worse, comparable, and better than the comp+rts version (that utilizes a hashmap), respectively. Table 7.5 shows the execution times and other stats of these benchmarks running baseline (no race detection), comp+rts, and treap on different input sizes. The execution times shown here are the average of three runs.

As the input size increases, the number of intervals n should increase as well, and we would like to understand how the overhead in treap may grow. Given a treap operation, its overhead is $O(h + k)$ where h is the height and k is the number of overlapping intervals. In the worst case, k can be large. The data in the two right-most columns, however, shows that

k is typically small, and the overhead per treap operation is dominated by the nodes visited (bounded by $O(h)$).

Given that the treap operation is dominated by the tree height, one would expect the operation overhead grows with $O(\lg m)$ with high probability, where m is the number of nodes in the tree. As such, in the worst case, the execution time of a benchmark can increase and grow with $O(n \lg n)$, where n is the number of intervals generated during execution. Fortunately, as the numbers on the left indicate, the race-detector overhead compared to the baseline (*base*) remains fairly stable across different input sizes. This may seem counterintuitive, but the numbers shown in *treap oh* offer a clue: the overhead incurred by the treap data structures is relatively small compared to the rest of the race-detector overhead such that even if the treap overhead grows, the race-detector overhead is still dominated by other operations. Consequently, the treap overhead is too small to have much impact on the final execution time. The only exception to this is `fft`, which does not work well with using treap as its access history due its execution characteristics as explained earlier.

Chapter 8

Asynchronous Access History

The optimizations that targets access history presented in Chapter 7 performs well on most of the benchmarks we examined. However, the race detector that employs those optimizations has to execute a computation serially. Consequently, an interesting direction of subsequent work is how to extend these optimizations to parallel race detectors. Recall that the optimized access history consists of memory accesses coalescing mechanisms and a treap data structure that allows for checking races among and updating access history at the granularity of intervals. The main problem with parallelizing the optimized access history is to handle concurrent accesses to the treap data structure while still maintaining efficiency. Most concurrent search tree provide correctness guarantees such as linearizability, but do not provide performance guarantee on each operation. In the worst case, the latency of treap operations is linear in the number of workers used in the parallel execution, which can lead to poor overall speedup. Fortunately, the overhead of managing treap-based access history is relatively small in practice, compared to the overhead incurred by coalescing contiguous memory accesses. To illustrate this fact, we measured the overhead that incurred by runtime coalescing[19] and treap management separately. In Table 8.1, the coalescing column is the

[19]The compile-time coalescing incurs no overhead in terms of the running time of programs.

execution time spent on performing runtime coalescing.The treap column shows the running times account for treap management, including race detection. By comparing these two types of overhead, we can show that runtime coalescing incurs much higher overhead than treap management does, except for `fft` due to the large number of intervals it has.

	coalescing	*treap*
`chol`	17.19	1.63
`fft`	83.65	442.21
`heat`	16.44	2.62
`mmul`	194.93	20.61
`sort`	10.90	1.61
`stra`	35.21	1.87
`straz`	46.62	4.00

Table 8.1: The total time (in seconds) each benchmark spent on performing runtime coalescing and uptating treap, respectively.

Unlike the treap data structure which is difficult to parallelize, the runtime coalescing mechanism applies directly to parallel race detector since the runtime coalescing is performed during a strand's execution separately by the executing worker. Given this observation, we propose an ***asynchronous*** access history scheme, to extend our optimizations of access history to parallel race detectors. We separate the workers into ***core workers*** and ***treap workers***. The core workers execute the computation and perform runtime coalescing of executed strands in parallel, while the treap workers collect the coalesced intervals and perform race detection with treaps in series. Consequently, this asynchronous access history requires no parallelization of the treap data structure. This chapter describes the challenge of designing such a scheme, the required additional data structure, and the race detection protocol that checks for race asynchronously (Sections 8.1 through 8.3).

We have implemented a parallel race detector for fork-join programs based on this asynchronous access history and empirically evaluated it against the state-of-the-art parallel race

detector that targets fork-join parallelism [97] (Section 8.4). The empirical results indicate that the asynchronous access history based race detector incurs lower overhead and performs better.

8.1 Synchronous vs. Asynchronous Access History

The race detection algorithms discussed in this dissertation so far uses a ***synchronous*** access history, which performs race detection and access history management upon each memory access, or at the end of each executed strand for our optimized access history described earlier. In this section, we explores another approach, namely ***asynchronous*** access history, in which a set of ***core workers*** perform the core computation and runtime coalescing. The race detection and treap management, however, are deferred until an assigned ***treap worker*** collect the generated intervals asynchronously.

Recall that to perform runtime coalescing, our optimized access history uses a single bit-hashmap (described in Section 7.1.2) to keep track of which memory locations are accessed during a strand's execution. We still rely on this idea to perform runtime coalescing in asynchronous access history, with the difference that each core worker now maintains a thread-local bit-hashmap respectively, instead of a global one. By doing this, we can easily avoid the conflicts between any concurrent bit-hashmap operations and apply the runtime coalescing to parallel race detector.

With the asynchronous access history, however, new challenges arise. Consider Figure 8.1 as an example. In this figure, strand b and c cause a read-write race. With synchronous access history, the race detector always race checks strand b and c before continuing executing strand d. In asynchronous access history, however, the core workers do not perform race detection while executing the core computation. If the asynchronous access history is simply implemented such that the treap workers collect available intervals of executed strands in an arbitrary order, the race detector no longer ensures the correctness. For example, the

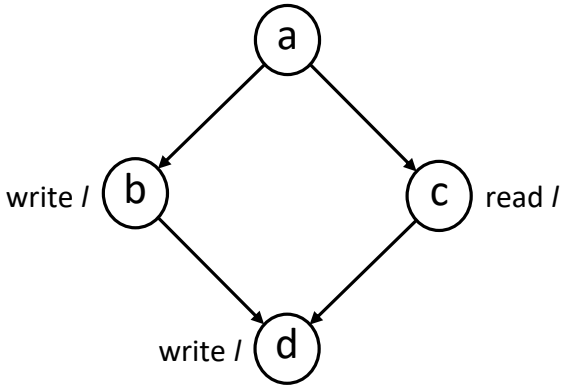


Figure 8.1: A parallel computation that contains a determinacy race. However, no race will be reported if we perform race detection in a certain order.

treap workers could collect executed strands and perform race detection asynchronously in the following order: $a \rightarrow c \rightarrow d \rightarrow b$. Recall that when race detecting parallel programs, only the last node that wrote to a given memory location (i.e., the last writer) is stored in the access history. In this sequence, d becomes the last writer of l upon collecting b and therefore no race is reported since $b \prec d$. In order to avoid such an issue, the treap workers must process intervals in a way that respects the dependences between strands.

The other challenge of the asynchronous scheme is clearing the memory accesses off the access history. Figure 8.2 shows an invocation tree of a task-parallel code and its corresponding stack shift when returning from spawned function B and calling E. As we call see, the stack frames of B and E may overlap in the stack of the executing worker and consequently, two logically distinct local variables x and y could potentially have the same memory address and thus cause a false positive. In synchronous access history, the race detector clears the corresponding stack accesses off the access history when exiting a stack frame to avoid reporting false races. However, we cannot simply follow the same approach in the asynchronous access history. When exiting the stack frame of function B, for example, there is no guarantee that the child strands of B (i.e. C and D) have been race checked by the treap workers. Consider the following situation. Say C and D have a conflicting memory

access on the variable allocated in B's stack frame. When exiting B's stack frame, the treap workers only race checked and updated C's accesses into the access history and not yet race checked D's accesses. If the tool clears the access history now, it will erase C's accesses to all the variable allocated in B's stack frame and therefore the tool will not catch a race when race checking on D.

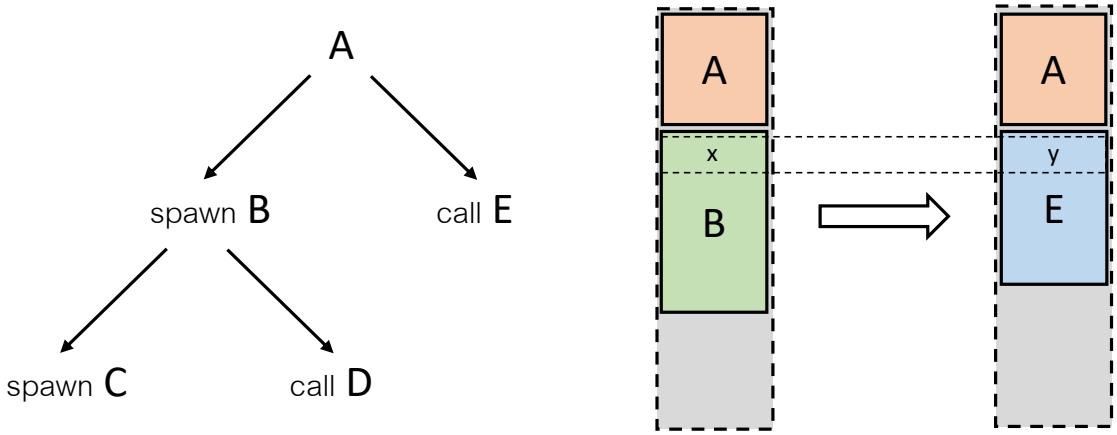


Figure 8.2: An invocation tree and its corresponding views of stack.

8.2 The Trace Data Structure

To address these challenges, we use an additional data structure called the ***trace***, to force dependences between strands to be respected when treap workers collect and race check the corresponding intervals. A trace stores a sequence of strands that executed by a core worker and it is ended under two situations:

1. The core worker hits a `sync` and the continuation strand of the corresponding `spawn` is stolen.

2. The core worker returns from a spawned task and realizes the continuation strand is stolen.

Within a trace, a core worker executes the strands sequentially as the serial execution, which follows left-to-right depth-first traversal order of the computation dag. Naturally, a trace has the following property: given two strands u and v in the same trace. If $u \prec v$, then u is before v in the trace.

Furthermore, each strand in the trace maintains additional bookkeeping data. For a given trace t, a strand s in t has three fields: $s.intvl$, $s.child$ and $s.nop$. Field $s.intvl$ points the set of intervals generated within s. Field $s.child$ points to the child strand of s. In the case that s has two child strands (i.e., s is a spawn node), $s.child$ always points to the continuation strand of s. Finally, field $s.nop$ indicates the number of parent (nop) strands that have not been race checked. The *child* and *nop* fields are crucial to ensure the correctness of the race detector that uses asynchronous access history, as we will discuss later in Section 8.3. Specifically, we say a trace t is ***ready*** if and only if the first strand of t has a *nop* field equals zero.

In the asynchronous race detector, each core worker maintains a *current* trace and a trace *pool*, that is a set of ended trace created by the core worker. Now we describe, relatively briefly, how to construct the trace data structure on-the-fly. Algorithm 14 shows the construction code. Upon executing a strand u, the construction algorithm always lets $u.child$ point to the child strand of u (omitted in the pseudocode). In particular, if u is a spawn node, the algorithm points $u.child$ to the continuation strand, set the *nop* field of the continuation strand and the corresponding sync strand to one and two, respectively (lines 3–5). If u is a sync node, the algorithm will decide whether to end the current trace or not, based on if the continuation strand of the corresponding spawn node is stolen (lines 8–10). Finally, when a core worker exits a spawned task, it checks if the continuation strand gets stolen. If so, the core worker ends the current trace and begins a new trace (lines 13–15).

After executing a strand, the algorithm will set the *intvl* fields to the set of intervals computed during the strand, put the executed strand at the end of the current trace of the executing worker.

Algorithm 14: Trace Construction

```
1  Function Spawn(u)
2      let s be the corresponding sync strand and c be the continuation strand, respectively
3      u.child = c
4      c.nop = 1
5      s.nop = 2
     /* w is the executing worker                                          */
6  Function Sync(u)
7      let v be the corresponding spawn node of u
8      if the continuation strand of v is stolen then
9          w.tracepool.put(w.current)
10         w.current = new Trace()
     /* w is the executing worker                                          */
11 Function TaskExit()
12     let c be the continuation strand
13     if c is stolen then
14         w.tracepool.put(w.current)
15         w.current = new Trace()
```

8.3 Asynchronous Race Detection Protocol

As mentioned previously, the asynchronous access history scheme separates the workers into ***core workers*** and ***treap workers***. In particular, a core worker performs the following activities:

- Executes the core computation.
- Maintains a bit-hashmap to keep track of the accessed memory locations within a strand and compute the intervals accessed after the strand finishes (described in Section 7.1.2).
- Maintains a trace data structure to track dependences between strands to ensure the correctness of race detection with asynchronous access history (described in Section 8.2).

In this section, we describe how the treap workers collect the intervals computed by the core workers and performs race detection. When race detecting a fork-join program in parallel and given a memory location, it suffices to store two readers, that is the left-most reader and the right-most reader ,and the latest writer for the correctness [55]. Similar to the interval-based access history described in Section 7.2, we use separate treaps to keep track of those readers and writer given an interval. We bring additional parallelism by assigning each treap to a single treap worker.

In order to guarantee the correctness of race detection, we force treap workers to only collect strands from a ready trace. In particular, each treap worker iterates through the traces maintained by the core workers, finds a ready trace, and collects strands one after another from the ready trace. The following lemma says that, by doing this, the dependences between strands are respected by the treap workers when collecting corresponding intervals.

Lemma 66. *Given two strands u and v. If $u \prec v$, then u must be collected before v by the treap workers.*

Proof. If u and v are in the same trace, then by the property of trace data structure, u is stored before v in the trace. Therefore u must be collected before v.

If u and v are in different traces, then there are two cases. The first case is that v is the first strand of its trace. Since a trace can only be collected when it's ready (i.e., $v.nop = 0$), u must be collected already.

The second case is that v is not the first strand of its trace. If v is neither a sync node nor a continuation node of a spawn node, then v must be executed by the same worker that executed its parent within the same trace. If v is either a sync node or a continuation node, then v must be in the trace that its parent resides as well. Otherwise v is the first strand of its trace due to stealing, which contradicts the assumption that v is not the first strand. Given these claims, one can apply induction to show that u and v must be in the same trace, which leads to a contradiction against the original assumption. □

Now take the write treap worker (shown in Algorithm 15), as an example. Once a ready trace is found, the write treap worker collects all the strands on that trace in sequence. For each strand, the write treap worker first checks write-write race against the write treap while updating the write treap (line 3). The write treap worker then checks read-write race for all the read intervals computed during that strand (line 4). Furthermore, after race checking a strand s, the treap workers decrement the *nop* field of $s.child$ if s and $s.child$ are in different traces (lines 5–6). To accommodate the concurrency that multiple treap workers collect strand simultaneously, extra copies of *nop* fields of a strand are required. For example, when the write treap worker collect a strand s, it decrements the nop_w fields of $s.child$. From the perspective of the writer treap worker, a trace is ready when $f.nop\_w = 0$, say f is the first strand of the trace.

Algorithm 15: Asynchronous Race Detection Protocol

```
   /* t is a ready trace                                              */
 1 Function WriteTreapWorker(t)
 2 |   foreach strand s ∈ t do
 3 |   |   CheckAndUpdateWriteTreap(s.intvl.write)
 4 |   |   CheckWriteTreap(s.intvl.read)
 5 |   |   if s.child ∉ t then
 6 |   |   |   s.child.nop_w = s.child.nop_w − 1
   /* t is a ready trace                                              */
 7 Function RightMostTreapWorker(t)
 8 |   foreach strand s ∈ t do
 9 |   |   CheckAndUpdateRightMostTreap(s.intvl.read)
10 |   |   CheckRightMostTreap(s.intvl.write)
11 |   |   if s.child ∉ t then
12 |   |   |   s.child.nop_rr = s.child.nop_rr − 1
   /* t is a ready trace                                              */
13 Function LeftMostTreapWorker(t)
14 |   foreach strand s ∈ t do
15 |   |   CheckAndUpdateLeftMostTreap(s.intvl.read)
16 |   |   CheckLeftMostTreap(s.intvl.write)
17 |   |   if s.child ∉ t then
18 |   |   |   s.child.nop_lr = s.child.nop_lr − 1
```

Now we look at how the trace data structure helps to address the second challenge of the asynchronous race detector, that is, how to clear the access history correctly. When exiting a stack frame f, the race detector simply put a special ***clear stack*** strand of this particular stack frame in the trace immediately after its current strand, without actually clearing the access history. Then after collecting this strand, the treap workers will clear all the accesses of stack f off the access history. Due to Lemma 66, we are guaranteed that all the child strands that may have accessed f are collected and race checked and there is no race missed. Also, since a trace serializes all strands executed by a single worker, the treap workers will collect a strand only after the accesses of the potential overlapped stack frame have been cleared and therefore no false positive will be reported by the tool.

The following theorem says that the asynchronous access history still guarantee the standard correctness of race detection algorithms.

Theorem 67. *Given a fork-join program, the race detector with asynchronous access history reports a race if and only if the program has a race.*

Proof. It is clear that the race detector reports a race only when finding overlapping intervals that have parallel accessors and one of the interval is a write, no matter what types of access history it uses. Also, the trace data structure guarantees that the access history is cleared correctly when needed. Therefore, the race detector reports no false races.

Now assuming the program has a pair of conflicting memory accesses on interval l – writer u and reader v. Say u is collected before v. When the write treap worker performs `CheckWriteTreap` on v, we must have $v \parallel \texttt{writer}(l)$ otherwise we have $u \prec v$ which contradicts $u \parallel v$. Therefore, a race will be reported. Note that Lemma 66 eliminates the situation that $v \prec \texttt{writer}(l)$.

If v is collected before u, then by applying the same argument in [55], we are guaranteed to have either $v \parallel \texttt{lreader}(l)$ or $v \parallel \mathit{rreader}(l)$. Thus, a race will be reported when the race detector performs either `CheckRightMostTreap` or `CheckLeftMostTreap`. □

8.4 Evaluation

To evaluate the efficiency of the asynchronous access history, we implement a prototype parallel race detector (referred to as ASYNC for short), that race detecting fork-join programs, using the WSP-Order algorithm [97] to maintain reachability, the compile-time and runtime coalescing scheme to compute intervals, and the asynchronous treap-based access history to perform race detection. We tested our race detector on six task-parallel benchmarks: Cholesky decomposition (`chol`, $n = 2000, z = 20000, b = 16$), parallel merge-sort (`sort`, $n = 2.5 \times 10^7, b = 2048$), heat diffusion simulation on a 2D grid (`heat`, $nx = 2048, ny = 2048, b = 10$), matrix multiplication (`mmul`, $n = 2048, b = 64$), Strassen's algorithm for matrix multiplication (two versions: `stra` and `straz` that uses Morton Z-layout, $n = 2048, b = 64$).[20]

We compare it to CRACER [97], the state-of-the-art parallel race detector for fork-join parallelism. CRACER uses an optimized hashmap to manage access history in synchronous fashion. To evaluate the impact of memory access coalescing and treap-based access history on the performance of a parallel race detector, we augmented CRACER with the compile-time and runtime coalescing mechanisms, but with the same hashmap to manage access history (referred to as CRACER+).

To make a detailed comparison, we compiled and tested four versions of the benchmarks:

- ***baseline:*** the original program without race detection;
- ***CRACER:*** the full race detection performed by CRACER;
- ***CRACER+:*** the full race detection performed by CRACER+; and
- ***ASYNC:*** the full race detection performed by ASYNC.

[20] We exclude fast-fourier transform (`fft`) benchmark in our evaluation because `fft` incurs significant overhead on managing treap-based access history, as shown in Section 7.3.

	baseline		*CRACER*		*CRACER+*		*ASYNC*	
	T_1	T_{20}	T_1	T_{20}	T_1	T_{20}	T_1	T_{20}
chol	0.61	0.04 [15.25×]	134.85 (221.06×)	7.47 [18.05×]	35.33 (57.91×)	2.94 [12.01×]	23.32 (38.22×)	2.48 [9.40×]
heat	4.37	0.40 [10.92×]	515.02 (117.85×)	29.88 [17.23×]	265.26 (60.70×)	17.64 [15.03×]	25.52 (5.83×)	2.16 [11.81×]
mmul	8.11	0.41 [19.78×]	414.10 (51.06×)	20.88 [19.83×]	242.54 (29.90×)	12.30 [19.71×]	245.77 (30.30×)	14.37 [17.10×]
sort	3.38	0.17 [19.88×]	99.94 (29.56×)	12.38 [8.07×]	74.75 (22.11×)	4.97 [15.04×]	23.34 (6.90×)	1.82 [12.82×]
stra	1.48	0.18 [8.22×]	583.57 (394.30×)	57.62 [10.12×]	137.33 (92.79×)	14.10 [9.73×]	41.38 (27.95×)	3.07 [13.47×]
straz	1.53	0.11 [13.90×]	323.78 (211.62×)	27.25 [11.88×]	155.63 (101.71×)	9.86 [15.78×]	74.35 (48.59×)	6.02 [12.35×]

Table 8.2: Execution times (in seconds) of different versions of the benchmarks. Columns with T_1 show the single-core execution times and columns with T_{20} show the 20-cores execution times. Numbers in the parentheses show the overhead compared to the baseline. Numbers in the brackets show the scalability compared to its respective single-core execution (T_1).

Overhead of ASYNC. The T_1 column in Table 8.2 shows the sequential running time for all versions of tested benchmarks. This comparison is somewhat similar to what we have done for the sequential treap-based access history in Section 7.3. The difference is that both CRACER and CRACER+ additionally use fine-grained locks to synchronize the query and update into the hashmap-based access history, which incurs additional overhead compared to the vanilla and comp+rts versions evaluated in Section 7.3. Also, during a sequential execution, ASYNC first performs the core computation as well as memory access coalescing, and then performs race detection and updates the treap one after the other. As shown in Table 8.2, ASYNC incurs higher overhead compared to the race detector augmented with memory access coalescing and treap that evaluated in Section 7.3. This is because ASYNC maintains additional trace data structure and employs extra reader treap in access history. By comparing CRACER and baseline, we can see the race detection and access history management incurs significant overhead. Due to the memory access coalescing mechanisms,

CRACER+ incurs much less overhead and the additional optimization enabled by the treap in ASYNC brings additional benefit.[21].

Scalability of ASYNC. We have also evaluated the performance of each race detector in parallel with 20 processor cores. When running on 20 cores, ASYNC employs 17 core workers to perform core computation and runtime coalescing, and 3 treap workers to manage the write treap and the reader treaps, respectively. The T_{20} column in Table 8.2 shows the parallel running time on 20 cores, and the number in the parentheses shows the scalability each version of benchmark achieves during parallel execution. For most benchmarks, the overhead reduction of the treap-based access history compared to hashmap retains in the parallel execution. In addition, both CRACER+ and ASYNC scales similarly to the baseline in general, as the number of processor cores increases, which provides a significant boost to the performance of race detection compared to CRACER.

[21]The only exception is `mmul`, which is explained in Section 7.3

Chapter 9

Related Work

In this chapter, we briefly review the related work on the studies presented in this dissertation.

Race detection for two-dimensional dags

Dimitrov et al. [26] provide the known algorithm for on-the-fly race detection for 2D-dags. This algorithm must execute the program sequentially and it has never been implemented and evaluated in practice. Their algorithm uses Tarjan's nearly linear-time least-common-ancestor (lca) algorithm [93] to identify the lowest-common-descendent of a pair of nodes in order to deduce whether they are in parallel. The use of Tarjan's lca algorithm leads to a small overhead in running time (functional inverse of Ackermann's function, which is small in practice — bounded above by 4).

Work-stealing runtime for synchronization primitives

Futures have been incorporated into many parallel platforms (e.g., [16–18, 32, 36, 47, 51, 88, 94]). Many use parsimonious work-stealing [17, 32, 36, 47, 88].

Other variants of work-stealing have also been implemented to support synchronization primitives that can cause suspension, but none of them provide provably efficient scheduling

bounds. In variants of X10 [18, 91] and Habanero dialects [16, 39], various synchronization primitives other than futures are provided that can cause the execution to block while the executing worker's deque is not empty. In the initial release of X10 [18], little support was provided — a blocking synchronization primitive blocks the executing worker, and to compensate, the runtime spawns a new worker thread to replace the blocked worker, effectively suspending the deque. Over time, the system could be oversubscribed. Tardieu et al. [91] subsequently developed a version of X10 with better support for suspension. In their system, if a worker is blocked the worker suspends the blocked task, but uses a centralized queue to allow resumptions of suspended tasks. A similar approach is taken by the initial release of Habanero Java [16]. In a later version, Imam and Sarkar [39] describe support for suspension in Habanero Java. In their system, suspended tasks are stored with the blocking synchronization primitives (similar to how we handle futures), but once the tasks get resumed, they all get pushed onto the ready deque of the worker who unblocks them.

Work-stealing analysis with multiple deques

Researchers have proposed provably efficient work-stealing schedulers where the execution allows for suspensions [57, 99]. Muller and Acar [57] studies a work-stealing scheduler that hides latency of I/O operations. When a worker encounters I/O, it may suspend the currently executing task. Their scheduler is parsimonious, but due to the possibility of suspension, there can be more than P number of deques in the system. Their scheduler provides a bound of $O(T_1/P + T_\infty U(1 + \lg U))$, where U is the maximum number of parallel I/O operations. Utterback et al. propose [99] a processor-oblivious record-replayer for fork-join parallel computations that utilize locks. During replay, if a lock-acquire is not "ready" to be replayed, the executing worker suspends its current deque and steals. Our performance bound analysis takes inspiration from theirs, but we need to additionally handle muggable deques. In their system, the number of suspended deques can be unbounded, and the

scheduler provides the time bound of $O(T_1/P + T_\infty \lg\lg P)$. Instead of randomly choose a victim to deposit the suspended deques, they utilize the power-of-two choices, choosing two victims and deposit it with the one with the lighter load, thereby obtaining a slightly better bound ($\lg\lg P$ in front of the T_∞ term instead of $\lg P$). We cannot apply the same strategy, since the power-of-two choices does not seem to help with bounding the minimum load [101].

Race detection for futures

A few algorithms [2, 90, 98] have been proposed to race detect programs that use futures. Surendran and Sarkar [90] proposed the first algorithm to race detect a program that uses futures, but their algorithm runs sequentially and can incur large overhead, $O(T_1(f+1)(k+1)\alpha(m,n))$, where f is the number of future tasks, k is the number of future operations, m is the number of memory accesses, n is the number of parallel control constructs executed, and α is the functional inverse of Ackermann's function. Later, Agrawal et al. [2] improved the bound to $O(T_1 + k^2)$, although the work is theoretical and no implementation of the algorithm exists. Finally, Utterback et al. [98] separated the use of futures into two classes — structured use of futures and general use of futures. By distinguishing the two, Utterback et al. [98] observed that programs that use structured futures can be race detected much more efficiently, in time $O(T_1\alpha(m,n))$. For general use of futures, Utterback et al. [98] proposed an algorithm (and its corresponding implementation) that executes in time $O((T_1+k^2)\alpha(m,n))$.

Optimizing access histories of race detectors

Many tools employ some kind of shadow memory to store shadow values with different memory locations in the program-under-test. Memory checkers, such as Valgrind [60], Dr. Memory [15], and AddressSanitizer [83] store metadata that shadows each byte of memory in the program-under-test to track which memory locations are safe or unsafe to access. Race detectors for pthreaded code such as Eraser [79] and ThreadSanitizer [84] store locksets and happens-before information as shadow values for locations in shared memory. The FastTrack

race detector [30] stores vector clocks and thread IDs as shadow values with each memory location.

Researchers have acknowledged the importance of efficient shadow-memory data structures for such tools [59] and have explored ways to optimize shadow memory data structures. One strategy is to employ different schemes to encode the shadow values in order to minimize the amount of data stored for each memory location [20, 59, 69, 79, 83, 84]. Such optimizations are tool specific and, in some cases, can affect the precision of the tool's analysis. Researchers have also explored different table structures for implementing the shadow memory. In the direct-addressing schemes [20, 66, 69, 83], the shadow memory is implemented simply as an array in memory, and each memory location in the program is mapped to a location in this array during a simple scale-and-offset computation. Such a scheme can be efficient, but the program must part with a fraction of virtual memory to utilize the tool. Another scheme is the multilevel translation schemes [59, 102], which provide flexibility in the allocation of the shadow memory but incur a higher cost to map memory locations to their shadow values. Researchers have also explored optimizations, such as vectorization, for accessing and updating the ranges of entries in the shadow memory table [59]. Our work employs coalescing and a tree data structure to store shadow values at the interval granularity. Even though the tree-based structure has worse theoretical performance than the table structures typically used, surprisingly, by exploiting coalescing, we show that the tree-based shadow memory can outperform the table-based one on many task-parallel code due to their data-access patterns.

Coalescing has been explored as an optimization in the context of data race detectors for pthreaded code. RedCard [31] and SlimFast [67] employ compile-time coalescing. Since these detectors still utilize a hashtable for shadow memory, however, they can coalesce together a set of variables only when the compiler can statically prove that they are accessed together in all synchronization-free regions (SFRs) so that a single key can be used to represent the set in

the hashtable. In contrast, our work finds memory locations to coalesce together dynamically at runtime. These locations can change throughout execution. Hence, we cannot use just one key to represent them in a hashtable, which would necessitate multiple queries per set.

SlimState [100] and BigFoot [77] employ dynamic coalescing for (only) arrays. These works convert an array into an object with multiple partitions, where each partition represents either a block of contiguous memory accessed in the array or a particular strided access pattern for part of the array. Access patterns outside of these predefined categories cause the implementation to revert back to fine-grained access tracking. Moreover, when a new SFR commits its accesses, if these accesses span multiple existing partitions in the shadow memory, one must perform checks against each of such overlapping partitions and refine the existing partitions to incorporate the new accesses. These papers do not detail what data structures are used to store such partitions and how much overhead such an operation incurs, as execution time bound is not their primary focus. Our work focuses on coalescing memory accesses that span contiguous memory locations (i.e., intervals); it is not limited to arrays with particular access patterns. Moreover, we show that our treap construction allows for provably efficient insertion or updating of a new interval.

Work by Park et al. [65] is perhaps the most closely related to our work. Their data-race detector employs a skiplist to manage shadow memory. However, a key difference is that their detector does not remove redundant intervals. If a new interval x overlaps existing intervals y and z, after insertion of x, all three intervals co-exist in the skiplist. Our work replaces the existing intervals upon insertion of x (though x is checked against y and z). Doing so allows the insertion, update, or query of a given interval to be done in time $O(\lg n + k)$, where k is the number of overlapping intervals in the data structure when the new interval is inserted. This efficiency is necessary for the final execution-time bound. Their bound for insertion, update, and query is $O(\lg^2 n + k)$ time, where k, the number of overlapping intervals, might

be much larger because such overlapping intervals may increase over time due to duplicate intervals.

Chapter 10

Conclusion

This dissertation has explored race detection problems outside of the realm of fork-join parallelism that has been extensively studied by the prior work [9, 27–29, 55, 71, 72, 97]. Specifically, this dissertation has presented the following race detection algorithms:

1. ***2D-Order*** that race detects programs use pipeline parallelism with asymptotically optimal running time (Chapter 3). 2D-Order exploits the order-dimension two property of 2D-dags that generated by the executions of pipeline programs and achieves efficient reachability algorithm.

2. ***F-Order*** that targets a more general class of parallel computation: futures (Chapter 5). It is the first known parallel race detection for general futures with provable performance guarantee. F-Order employs a novel data structure that answers reachability query efficiently, despite the lack of structural properties of NSP-dags that generated by executing task-parallel programs using futures.

3. ***SF-Order*** that performs race detection on programs with the structured use of futures (Chapter 6). By exploiting the restrictions imposed by structured futures, SF-Order

achieves asymptotically more efficient time bound compared to F-Order, and incurs lower overhead in practice.

All the race detection algorithms discussed above primarily focus on designing efficient reachability data structure and use an optimized two-level page-table-like hashmap to manage access history. Such access history incurs constant overhead per query and update in theory but significant overhead in practice. Take the empirical results from the experimental evaluation of SF-Order algorithm, as an example (Section 6.4). The performance gain of SF-Order against F-order is not significant even though SF-Order is asymptotically more efficient because the overhead of race detection is dominated by the access history management. Therefore, this dissertation has also investigated different approach of redesigning the access history. Specifically, this dissertation has presented the following optimizations on access history to offer additional boost to the overall performance of race detectors:

1. compile-time and runtime memory access coalescing mechanisms and a treap-based access history data structure that speed up sequential race detectors for fork-join code (Chapter 7).

2. an asynchronous access history scheme that extends those optimizations to parallel race detectors (Chapter 8).

Finally, during the research of race detection problem for futures, we found the programs with futures could incur much higher scheduling costs when scheduling with the classic work-stealing algorithm. To build provably and practically efficient runtime system for futures, we propose an alternative scheduling approach – ***proactive work-stealing*** (Chapter 4). ProWS provides equal or better bounds on the number of deviations compared to the classic work-stealing algorithm.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–12, 2000.

[2] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. Race detection and reachability in nearly series-parallel dags. In *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2018.

[3] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010.

[4] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, 1998.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, pages 115–144, 2001.

[6] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, 1977.

[7] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 735–736, 2009.

[8] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[9] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 133–144, 2004.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[11] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 249–259, 1997.

[12] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, July 1995.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 356–368, November 1994.

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[15] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223, 2011.

[16] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 51–61, 2011.

[17] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[18] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[20] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC)*, pages 749–754, 2006.

[21] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, August 1988.

[22] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: a DSL approach to specifying streaming applications. In *Generative Programming and Component Engineering (GPCE)*, pages 1–17, 2003.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[24] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, December 2008.

[25] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, May 1987.

[26] Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. Race detection in two dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 101–110, 2015.

[27] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, June 1997.

[28] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[29] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[30] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *ACM SIGPLAN Notices*, 44(6):121–133, June 2009.

[31] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant check elimination for dynamic race detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, pages 255–280, July 2013.

[32] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, 20(5-6):537–576, November 2010.

[33] D.P. Friedman and D.S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, 1978.

[34] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[35] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[36] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[37] Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 155–166, 2014.

[38] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 121–130, 2010.

[39] Shams Imam and Vivek Sarkar. Cooperative scheduling of parallel tasks with general synchronization patterns. In *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, pages 618–643, 2014.

[40] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[41] Intel® Cilk™ Plus. `https://www.cilkplus.org`, 2013.

[42] Intel Corporation. Intel® Cilk™Plus. Available from `https://www.cilkplus.org/`, 2011. Accessed: August 2017.

[43] Intel Corporation. *Intel(R) Threading Building Blocks*, 2012. Available from `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm`.

[44] Intel Corporation. *Intel® Cilk™ Plus Language Extension Specification, Version 1.1*, 2013. Document 324396-002US. Available from `http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm`.

[45] Intel Corporation. Piper: Experimental language support for pipeline parallelism in Intel® Cilk™Plus. Available from `https://www.cilkplus.org/piper-experimental-language-support-pipeline-parallelism-intel-cilk-plus`, 2013. Accessed: August 2017.

[46] ISO/IEC 14882:2011(e) information technology — programming languages — c++, 2012. Third Edition, 2012-02-14.

[47] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *The ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 81–90, 1989.

[48] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 140–151, July 2013.

[49] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing*, 2(3):17:1–17:42, September 2015.

[50] Charles E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3):244–257, 2010.

[51] Li Lu, Weixing Ji, and Michael L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–529, 2014.

[52] S. MacDonald, D. Szafron, and J. Schaeffer. Rethinking the pipeline as object-oriented states with transformations. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, pages 12–21, 2004.

[53] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. In *ACM SIGGRAPH*, pages 896–907, 2003.

[54] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[55] John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing*, pages 24–33, 1991.

[56] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge University Press, 2nd edition, 2017.

[57] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 71–82, 2016.

[58] Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. Responsive parallelism with futures and state. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 577–591, June 2020.

[59] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE)*, pages 65–74, 2007.

[60] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[61] Robert H. B. Netzer and Barton P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[62] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[63] *OpenMP Application Program Interface, Version 4.0*, July 2013.

[64] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *The IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[65] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[66] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference (USENIX ATC)*, pages 115–126, June 2013.

[67] Yuanfeng Peng, Christian DeLozier, Ariel Eizenberg, William Mansky, and Joseph Devietti. SLIMFAST: Reducing metadata redundancy in sound and complete dynamic data race detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 835–844, 2018.

[68] Antoniu Pop and Albert Cohen. A stream-computing extension to OpenMP. In *High-Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 5–14, 2011.

[69] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006.

[70] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–123, 2008.

[71] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. In *International Conference on Runtime Verification*, pages 368–383. 2010.

[72] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 531–542, 2012.

[73] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 177–188, 2004.

[74] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th International Conference on Supercomputing (ICS)*, pages 137–146, 1995.

[75] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming*, 23(6):537–576, December 1995.

[76] Lawrence Rauchwerger and David A. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, February 1999.

[77] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. Bigfoot: Static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–156, 2017.

[78] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 22–32, 2011.

[79] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[80] Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. Brief announcement: Open Cilk. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 351–353, 2018.

[81] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding recursive fork-join parallelism into llvm's intermediate representation. *ACM Transactions on Parallel Computing (TOPC)*, 6(4), December 2019.

[82] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. In *ALGORITHMICA*, pages 540–545, 1996.

[83] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[84] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, pages 62–71, 2009.

[85] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. Priority scheduling for interactive applications. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 465–477, July 2020.

[86] Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. Proactive work stealing for futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–271, 2019.

[87] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 91–100, 2009.

[88] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 253–264, 2008.

[89] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed pipeline parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 147–156, 2010.

[90] Rishi Surendran and Vivek Sarkar. *Dynamic Determinacy Race Detection for Task Parallelism with Futures*, pages 368–385. 2016.

[91] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 267–276, 2012.

[92] Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. Zip trees, 2018.

[93] Robert Endre Tarjan. Applications of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, October 1979.

[94] Sağnak Taşırlar and Vivek Sarkar. Data-driven tasks and their implementation. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP)*, pages 652–661, 2011.

[95] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *The IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.

[96] Robert Utterback. `https://github.com/wustl-pctg/futurerd`, 2019. Accessed in August 2019.

[97] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 83–94, 2016.

[98] Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. Efficient race detection with futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 340–354, 2019.

[99] Robert Utterback, Kunal Agrawal, I-Ting Angelina Lee, and Milind Kulkarni. Processor-oblivious record and replay. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 145–161, 2017.

[100] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. Array shadow state compression for precise dynamic race detection. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 155–165, 2015.

[101] Weiyu Xu and A. Kevin Tang. A generalized coupon collector problem. *Journal of Applied Probability*, 48(4):1081–1094, 2011.

[102] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 22–31, 2010.